

Swansea University E-Theses

Hardware accelerated volume texturing.

Miller, Christopher Michael

How to cite:

Miller, Christopher Michael (2006) *Hardware accelerated volume texturing.* thesis, Swansea University.
<http://cronfa.swan.ac.uk/Record/cronfa42524>

Use policy:

This item is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence: copies of full text items may be used or reproduced in any format or medium, without prior permission for personal research or study, educational or non-commercial purposes only. The copyright for any work remains with the original author unless otherwise specified. The full-text must not be sold in any format or medium without the formal permission of the copyright holder. Permission for multiple reproductions should be obtained from the original author.

Authors are personally responsible for adhering to copyright and publisher restrictions when uploading content to the repository.

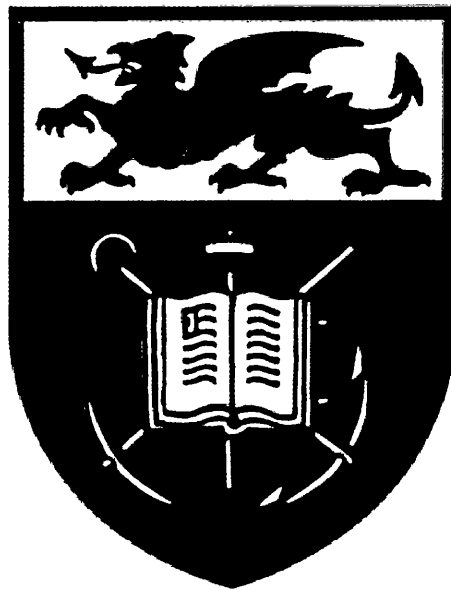
Please link to the metadata record in the Swansea University repository, Cronfa (link given in the citation reference above.)

<http://www.swansea.ac.uk/library/researchsupport/ris-support/>

Hardware Accelerated Volume Texturing

Christopher Michael Miller BSc. (Wales)

A thesis submitted to the University of Wales in
candidature for the degree of Philosophiae Doctor



Department of Computer Science
University of Wales, Swansea

September 2006



ProQuest Number: 10805273

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10805273

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed (candidate)

Date 15/03/07

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed (candidate)

Date 15/03/07

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed (candidate)

Date 15/03/07

Summary

The emergence of *volume graphics*, a sub field in computer graphics, has been evident for the last 15 years. Growing from scientific visualization problems, volume graphics has established itself as an important field in general computer graphics. However, the general graphics fraternity still favour the established surface graphics techniques. This is due to well founded and established techniques and a complete pipeline through software onto display hardware. This enables real-time applications to be constructed with ease and used by a wide range of end users due to the readily available graphics hardware adopted by many computer manufacturers. Volume graphics has traditionally been restricted to high-end systems due to the complexity involved with rendering volume datasets. Either specialised graphics hardware or powerful computers were required to generate images, many of these not in real-time.

Although there have been specialised hardware solutions to the volume rendering problem, the adoption of the volume dataset as a primitive relies on end-users with commodity hardware being able to display images at interactive rates. The recent emergence of programmable consumer level graphics hardware is now allowing these platforms to compute volume rendering at interactive rates. Most of the work in this field is directed towards scientific visualisation.

The work in this thesis addresses the issues in providing real-time volume graphics techniques to the general graphics community using commodity graphics hardware. Real-time texturing of volumetric data is explored as an important set of techniques in delivering volume datasets as a general graphics primitive.

The main contributions of this work are:

- The introduction of efficient acceleration techniques
- Interactive display of amorphous phenomena modelled outside an object defined in a volume dataset
- Interactive procedural texture synthesis for volume data
- 2D texturing techniques and extensions for volume data in real-time
- A flexible surface detail mapping algorithm that removes many previous restrictions

Parts of this work have been presented at the 4th *International Workshop on Volume Graphics* and also published in *Volume Graphics 2005*.

Acknowledgments

First and foremost I would like to thank my parents, without their support and enthusiasm throughout the years of my university education I don't believe that I would have had the drive and ambition to start or complete a PhD. I also extend my thanks to my girlfriend Laura who has put up with more than most would endure during the writing of this thesis. I would like to acknowledge her ability to spell words correctly, a skill that most of the time evades me. I have an overwhelming gratitude for their patience and encouragement whilst writing this thesis since research can take over a person and make them non-responsive to most of their surroundings at times. I would also like to thank Mr Crosby Chacksfield from my former sixth form college; his help, concern and enthusiasm whilst there helped shape my path through academia.

I would like to thank my supervisor Dr Mark Jones who constantly provided encouragement, insightful discussion, focus and direction wherever he could. I believe I have learned much from our discussions. In addition his overwhelming knowledge of the subject area was a constant source of inspiration and amazement.

My thanks also go to all my colleagues in the visual and interactive computing group and computer science department. All at some point have provided fruitful discussions around the groups academic pursuits, help and encouragement or more often, entertaining conversation. I would especially like to thank Prof Min Chen who has been a source of inspiration and encouragement throughout my time at Swansea. I finally thank my laboratory colleagues in particular for putting up with my many questions.

Contents

1	Introduction	1
1.1	Thesis Objectives	2
1.2	Thesis Outline	2
1.3	Featured Datasets	3
2	Volume Rendering	5
2.1	Volumetric Datasets	6
2.1.1	Computed Tomography	7
2.1.2	Magnetic Resonance Imaging	8
2.1.3	Functional Representation	8
2.1.4	Data Representation	9
2.2	Signal Reconstruction Filters	10
2.2.1	Nearest Neighbour	11
2.2.2	Trilinear Interpolation	11
2.2.3	Tricubic Interpolation	12
2.3	Volume Data Classification	13
2.3.1	Pre-Classification	15
2.3.2	Post-Classification	15
2.4	Gradient Computation and Shading	16
2.4.1	Gradient Computation	17
2.4.2	Shading	18
2.5	Iso-Surface Reconstruction	20
2.5.1	Contour Tracking	20
2.5.2	Marching Cubes	21
2.5.3	Marching Tetrahedra	23
2.5.4	Iso-Surface Tracking	23
2.5.5	Octree Encoding	24
2.6	Direct Volume Rendering	25
2.6.1	Image-Order Algorithm	28
2.6.2	Object-Order Algorithm	31
2.6.3	Hybrid Approaches	33
2.7	Volume Graphics	33
2.7.1	Distance Fields	34
2.7.2	Global Illumination	35
2.7.3	Shadows	36

2.7.4	Constructive Volume Representation	36
2.7.5	Deformation and Animation	37
2.7.6	Procedural Texture	39
2.7.7	Hypertexture	42
2.7.8	Texture Mapping	45
2.7.9	Bump Mapping and Displacement Mapping	46
2.8	Summary	47
3	Direct Volume Rendering	49
3.1	Hardware Pipeline	50
3.1.1	GPU Generations	52
3.1.2	Data Types	53
3.1.3	Memory and Registers	54
3.1.4	Vertex Processing	56
3.1.5	Fragment Processing	56
3.1.6	Texturing and Buffers	58
3.1.7	Branching	60
3.2	GPU Volume Rendering Algorithms	60
3.2.1	Object-order Proxy Slice Rendering Using Volume Textures	62
3.2.2	Object-order Proxy Slice Rendering Using 2D Textured Slices	65
3.2.3	Ray Casting	67
3.2.4	Distance Field Rendering	69
3.3	Improvements	73
3.3.1	Clipping	73
3.3.2	Signal Reconstruction	74
3.3.3	Pre-Integrated Classification	75
3.4	Comparison	80
3.4.1	OOP Rendering Framework	83
3.4.2	OOP Fuzzy Segmentation	85
3.4.3	OOP Binary Segmentation	86
3.4.4	OOP Results	87
3.4.5	IOM Rendering Framework	94
3.4.6	IOM Fuzzy Segmentation	98
3.4.7	IOM Binary Segmentation	99
3.4.8	IOM Results	101
3.4.9	IOS Rendering Framework	103
3.4.10	IOS Fuzzy Segmentation	104
3.4.11	IOS Binary Segmentation	105
3.4.12	IOS Rendering Results	106
3.5	Summary	109
4	Volume Texture and Hypertexture	111
4.1	GPU Procedural Texture Primitives	112
4.1.1	Pre-computed Evaluation	113
4.1.2	Local Evaluation	116
4.1.3	Results	117

4.2	Solid Texturing Volumetric Objects	120
4.2.1	Object-Order Proxy Slice Solid Texturing	122
4.2.2	Image-Order Single Pass Solid Texturing	123
4.2.3	Results	124
4.3	Volume Hypertexture	131
4.3.1	Distance Fields	132
4.3.2	DMF Functions	133
4.3.3	Example Hypertextures	135
4.3.4	Pre-Integrated Transfer Functions	136
4.3.5	Object-Order Proxy Slice Hypertexture	138
4.3.6	Image-Order Single Pass Hypertexture	139
4.3.7	Results	141
4.4	Animation Techniques	144
4.4.1	Texture Domain	148
4.4.2	Higher-Order Noise Primitives	149
4.4.3	Hypertexture Parameters	150
4.5	Summary	151
5	Volume Surface Detail	152
5.1	2D Texture Mapping	153
5.1.1	Forward Mapping	154
5.1.2	Backward Mapping	157
5.1.3	Intermediate Parametric Surfaces	158
5.1.4	Object-Order Proxy Slice 2D Texturing	160
5.1.5	Image-order Single Pass 2D Texturing	161
5.1.6	Results	163
5.2	Tangent Space	163
5.3	Bump Mapping	168
5.3.1	3D Bump Mapping	170
5.3.2	3D Bump Mapping Results	172
5.3.3	2D Bump mapping	176
5.3.4	2D Bump Mapping Results	179
5.4	Displacement Mapping	183
5.4.1	2D Displacement Mapping	185
5.4.2	2D Displacement Mapping Results	187
5.4.3	Volume Displacement Mapping	187
5.4.4	Volume Displacement Mapping Results	191
5.5	Summary	196
6	Conclusion	197
6.1	Achievements	198
6.2	Further Work	198

Chapter 1

Introduction

Contents

1.1	Thesis Objectives	2
1.2	Thesis Outline	2
1.3	Featured Datasets	3

Imagery is a long established means of communicating information between individuals and communities. Its use has evolved over many thousands of years from primitive manual techniques to fully automated systems. The computer has played an important role in the imagery that now surrounds us day to day. Artwork, memories, entertainment, knowledge and understanding have all benefited from computer graphics. Recent developments have also allowed such mediums to be conveyed on tiny mobile phone screens amongst other high-definition displays that only a few years ago would have been thought of as impossible. People now rely on the computer and its ability to display information as a part of every day life in a wide variety of disciplines including science, engineering, medicine, business, industry amongst many others.

Computer graphics is thus an important subject in many areas, most notably visualisation and entertainment. Both the visualisation and entertainment communities have made it possible to utilise the emerging programmable consumer level graphics hardware to extract images from volume datasets. The visualisation community has focused on methods to extract meaningful information from volumetric sources and provided many techniques to achieve this goal. The entertainment industry has driven the development of consumer level graphics hardware to enable end users to use an interactive environment for their entertainment and additionally developed many techniques to model real world situations with a computer for use in feature films and animations.

The majority of these products and features use the triangle as a modelling primitive. Surfaces are constructed with triangles to represent objects and surrounding medium. The surface and triangle however is not well suited to all naturally occurring mediums which the graphics community would like to capture and model in some way. A good example of the surface's inability to model naturally occurring mediums is fire. Fire does not have a well defined surface, and therefore the inclusion of such a medium is not well suited to the

surface modelling domain. In addition surface based techniques model objects as having an infinitely thin surface and no internal detail, which in the real world is far from reality.

Volume graphics has grown out of the need to enable such modelling behaviour intuitively and has grown largely from the medical imaging and visualisation communities. The volume approach encodes the physical properties of an object, its internal medium, its external surroundings and the complete definition of its surface without having to include complex techniques to simulate such properties. This is achieved by representing the densities in its defined space (e.g. 3D space). This volume graphics representation of density or matter in graphics is the foundation of this thesis.

1.1 Thesis Objectives

The main objective and focus of this thesis advance important volume graphics techniques to an interactive level on commodity hardware, an important goal in providing volumetric approaches to graphics to a wider audience and user base. Numerous researchers have noted the importance of volume graphics as a general primitive and have even suggested that the volume representations will eventually supersede the surface representation counterparts. This is due to surface representations exhibiting a complexity that grows with the models detail. Therefore rendering and manipulation complexity are bound by the detail present in the surface construction. Volume data can be rendered and manipulated with a known upper bound for the volume datasets size.

Volumetric approaches have already been proven to exhibit a superset of achievable intuitive modelling techniques, amongst other important properties such as a similar complexity for increasingly complex objects. However, it is usually the case that research is either based on the modelling of objects, or the rendering of objects and interactive display has been limited to visualisation problems above general graphics problems.

An objective for this work is therefore to provide an environment where rendering is achieved at interactive rates, and make possible the intuitive control of the detailed visual appearance of underlying objects. In addition the consideration of both the rendering and modelling disciplines should be considered with an appropriate framework definition to enable re-use of the work presented to a wider audience.

1.2 Thesis Outline

The work in this thesis is divided amongst four main chapters. Chapter 2 will introduce the areas of volume visualization and volume rendering and formalise the fundamental properties and techniques encountered within volumetric approaches. An in depth review of previous work from the subjects inception to state of the art research will be presented.

Chapter 3 introduces the underlying hardware implementations available on current commodity graphics hardware and provide a detailed review of existing techniques and strate-

gies. Additionally hybrids of these techniques will be investigated quantitatively and qualitatively to provide a benchmark in current interactive volume display.

Chapter 4 will explore procedural texture synthesis for volume data and volume hypertexture, a means of modelling and deforming volume datasets to closely model naturally occurring phenomena. Details of animation strategies and interactive computation of procedural textures will be presented.

Chapter 5 will demonstrate interactive texturing and fine surface detail methods for volume objects to enhance visual appearance without a large burden on modelling or rendering.

1.3 Featured Datasets

Table 1.1 provides a brief overview of the volumetric datasets used in this thesis. Each dataset is well known and it is for this purpose that they are used for testing and evaluation of rendering algorithms.

The *BuckyBall* dataset is synthesised and named after Buckminster Fuller. Bucky balls consist of 60 points on the surface of a spherical shape where the distance from any point to its nearest neighbouring three points on the sphere is identical.

The *AVSHydrogen* is a synthetic dataset describing electron distribution in a hydrogen atom.

The *CTHead* dataset is taken from a CT scanner during a head survey of a cadaver by the North Carolina Memorial Hospital.

The *CTHeadDist* dataset is a distance field encoding of the original *CTHead* dataset with the iso-surface being chosen as the bone structure or skull.

Both the sphere distance fields were generated directly from the parametric definition of a sphere for use in this work. The *SphereDist* dataset is very accurate and produces a smooth surface when rendered without under-sampling. The *SphereMeso* dataset is several sphere functions evaluated over a grid. These spheres are used to represent finite surface detail.

The frame rates reported in this thesis are based on a computer system equipped with a Pentium 4 2GHZ processor, 2GB of RAM and a NVIDIA 6800 graphics card with 256MB of video memory.

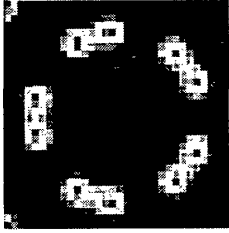
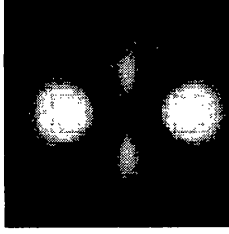

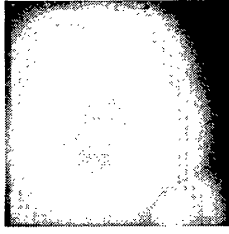
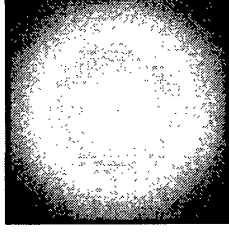
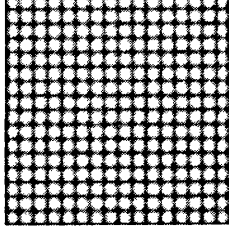
Image	Description
	<p>Name: <i>BuckyBall</i> Size: $32 \times 32 \times 32$ Scale: 1:1:1 Data type: 8 bit Source: AVS, USA</p>
	<p>Name: <i>AVSHydrogen</i> Size: $64 \times 64 \times 64$ Scale: 1:1:1 Data Type: 8 bit Source: AVS, USA</p>
	<p>Name: <i>CTHead</i> Size: $256 \times 256 \times 113$ Scale: 1:1:2 Data Type: 16 bit Source: UNC Chapel Hill</p>
	<p>Name: <i>CTHeadDist</i> Size: $256 \times 256 \times 128$ Scale: 1:1:2 Data Type: 32 bit Source: M.W. Jones, Swansea</p>
	<p>Name: <i>SphereDist</i> Size: $256 \times 256 \times 256$ Scale: 1:1:1 Data Type: 32 bit Source: C.M. Miller, Swansea</p>
	<p>Name: <i>SphereMeso</i> Size: $256 \times 256 \times 16$ Scale: 1:1:1 Data Type: 32 bit Source: C.M. Miller, Swansea</p>

Table 1.1: Featured Datasets

Chapter 2

Volume Rendering

Contents

2.1	Volumetric Datasets	6
2.2	Signal Reconstruction Filters	10
2.3	Volume Data Classification	13
2.4	Gradient Computation and Shading	16
2.5	Iso-Surface Reconstruction	20
2.6	Direct Volume Rendering	25
2.7	Volume Graphics	33
2.8	Summary	47

Volume visualisation has grown largely from medical diagnostic and teaching problems that require a method of conveying information about complex structures within the body. These structures are inherently three dimensional in nature and traditionally medical personnel have worked with two dimensional images to diagnose and learn about the human body. These two dimensional images could be photographs of cadavers or images obtained through *tomography* on live tissue.

Common tomographic techniques include *computed tomography* scans (CT), *magnetic resonance imaging* (MRI) and *positron emission tomography* (PET). These techniques facilitate taking a cross-section from a subject and include data for all of the internal densities encountered through a scan plane. Radiologists study these cross-sections to acquire an understanding of the three dimensional structure that a series of cross-sectional scans contain. However this understanding by trained individuals is difficult to convey to other doctors, students and patients.

Whilst two dimensional images obtained from photographs of cadavers or scans from live tissue can convey enough information for certain diagnosis and understanding, three dimensional representations are required to fully understand and explore complex diagnosis cases and medical science. Intuitive understanding across several disciplines is achieved by being able to generate a better visualisation that presents all of the information without having to reconstruct a picture mentally.

CT and MRI scanners can obtain a stack of two dimensional images, that when stacked provide a three dimensional cube of data. These three dimensional stacks or cubes of data define the notion of a volume dataset. A volume is therefore a three dimensional entity with external and internal detail of a particular subject. The graphics community predominantly concentrates on surface based representations for three dimensional images, however surfaces are incapable of defining an objects internal structure and external features intuitively. There are many applications and problems in the medical community exhibiting a volumetric nature. There are also numerous applications and problems across a diverse range of subject matter that require a volume visualisation approach. These include medical imaging for diagnostics, non-invasive simulation and training applications where volume data can be ascertained from scanning equipment. Industrial applications include non-destructive testing, reverse engineering, rapid prototyping and interactive modelling of 3D structures, products and entities.

This review chapter charts the early volumetric visualisation techniques from the inception of the subject, and major developments up to state of the art techniques. Volumetric dataset acquisition and data representation are explored in section 2.1. Extracting the desired information from a volume dataset is explored in sections 2.2, 2.3 and 2.4. A complete review of different approaches to obtaining a visualisation from a volume dataset is provided in sections 2.5 and 2.6. Finally the notion of *Volume Graphics* is considered as a general rendering primitive in section 2.7.

2.1 Volumetric Datasets

Since volume visualisation has grown mainly from medical imaging problems, most of the widely adopted datasets for exploring volumetric approaches are from the medical domain. CT and MRI scanners are discussed as a means to extract a volume dataset from live tissue or a cadaver. These techniques have also been used to acquire volume datasets of inert objects such as engine blocks. The Visible Human Project (VHP) was a national library of medicine [Pro] initiative to capture a male and female volume dataset of the whole body to use in research and teaching. The VHP includes CT and MRI scans of the entire body, and includes digital colour photographs of cross sections of the entire cadaver (see Figure 2.3(c)).

Volume datasets can be categorized as exhibiting different topologic and geometric properties [SK90]. The *CTHead* is an example of a volume dataset geometrically defined on a regular grid and topographically defined as structured. Other methods of volume acquisition can produce elements that are addressed geometrically on a non uniform grid and elements that are topographically defined on an unstructured grid (see Figure 2.1). Non uniform and unstructured grids are not considered as the focus of this work since it concentrates on uniform rectilinear grids.

Three dimensional data can be topologically defined as being:

- *Structured* - There exists a connectivity between each cell such that they can be addressed implicitly using topological co-ordinates.
- *Unstructured* - The connectivity between each cell must be represented explicitly.

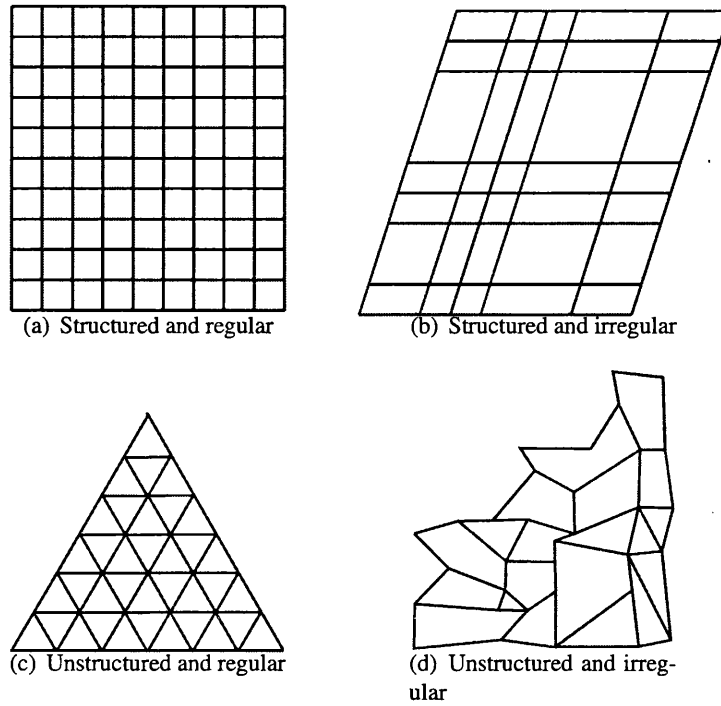


Figure 2.1: Differing grid geometries and topologies for volume data

Three dimensional data can be additionally defined geometrically as being:

- *Regular* - All cells in the grid are the same size and shape
- *Irregular* - The cells in the grid can exhibit different shapes and sizes from one another.

2.1.1 Computed Tomography

Computed Tomography (CT) or Computed Axial Tomography (CAT) is the acquisition of a planar 2D image from a subject by calculating x-ray absorption. The CT method was originally developed by Carmac and Hounsfield independently [Hou72]. X-rays are fired into the subject in a parallel, fan or cone shape from a transmitter. The absorption along these rays is recorded by a receiver. Rotating the scan plane through 360° over a fixed origin yields absorption line integrals that can be constructed into a 2D function $\mu(x, y)$ (see Figure 2.3(a)). Modern CT scanning equipment is shown in Figure 2.2.

The absorption integral function μ is measured in respect of the rays source intensity I_0 and distance between the emitter and receiver d :

$$I(x) = I_0 e^{-\int \mu(x) dx} \quad (2.1)$$

CT scanners become more precise over time by firing more x-rays to gain better resolution and minimizing scan time and radiation dosage. CT scanning is a non-invasive technique,

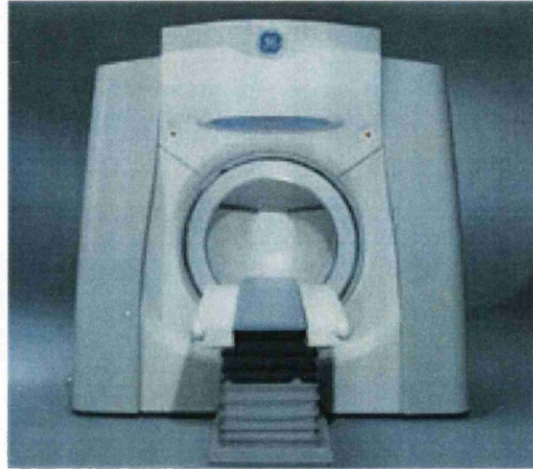


Figure 2.2: Modern CT Scanner, GE eSpeed EBT [Hea]

and is suitable for use on live human subjects in medical imaging. CT scanners detect boundaries between differing densities such as skin and bone effectively. However CT does not detect subtle variations in soft tissue. CT scanners are also widely adopted in the manufacturing industry to test and diagnose internal structures. 2D axial CT scans are stacked on top of one another to construct a volume dataset.

2.1.2 Magnetic Resonance Imaging

Magnetic Resonance Imaging (MRI) is a non-invasive form of scanning similar to CT. MRI detects subtle differences in soft tissue. MRI scanning builds on Nuclear Magnetic Resonance (NMR) developed independently by Purcel *et al.* [PTP45] and Blotch *et al.* [BHP46].

MRI scans are obtained by surrounding the subject with a strong magnetic field, which affects the orientation of protons, causing them to align with the magnetic field. A series of pulses are then delivered through the body at intervals to disturb the orientation of protons by resonance. A detector measures the orientation at each interval which enables the calculation of *proton density* (PD), *spin-lattice relaxation time* (T_1) and *spin-spin relaxation time* (T_2) where relaxation time measures the amount of orientation relaxed towards normal over time. These differing calculations represent three different types of MRI scan available. The resulting calculations are scaled to obtain a 2D axial images (see Figure 2.3(b)). Analogous to CT scanning, MRI axial images are stacked on top of one another to form a volume dataset.

2.1.3 Functional Representation

Functional Representation (*f-rep*) is a field in volume visualisation that encompasses continuous scalar field functions defined in three dimensions. Often scientific visualisation is modelled using functions rather than discretised data gathered from source materials using

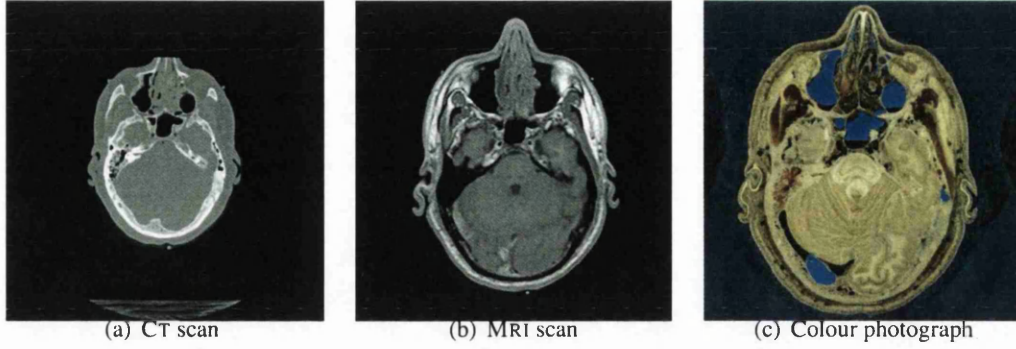


Figure 2.3: Axial images of CT, MRI and colour cryosections of the head

scanning techniques. Evaluation of these functions is done in the same manner as discretised data during rendering, however the scalar field is sampled at a lower resolution than the infinite resolution available with a functional representation. It is possible to voxelise a functional representation into a volume dataset as a pre-processing step.

There are many applications of functional representation. Computation Fluid Dynamics (CFD) is the study viscous fluids, non-viscous fluids and air movement around mediums in 3D space. These simulations encompass the necessary physical models to describe such flows and model several properties such as temperature, rate of flow and flow direction while accounting for forces such as friction and gravity. Volume visualisation methods are often used because these functions on their own provide little insight. The data captured is typically 3D with another possible dimension describing time (4D). Visualisations are generated to study flows, or a particular property at one time. For example when modelling air flow over a car, it might be desirable to visualize the speed of air over the car body. Using different colours to represent air speed such as green for fast moving and red for slow moving, an intuitive image can be ascertained.

Other applications for *f-rep* techniques include time-varying data, seismic data and scientific simulation (such as the *AVSHydrogen* dataset). Pasko *et al.* [PASS95] provide an review of *f-rep* techniques and applications.

2.1.4 Data Representation

The acquisition of a volume dataset via the methods defined above will produce volume datasets exhibiting a set of elements defined on a rectilinear grid (see Eqn 2.2).

A volume dataset, \mathbb{V} , is formally defined:

$$\mathbb{V} = \{v_i(x, y, z) \mid i = 1, 2, \dots, n\} \quad (2.2)$$

where $(x, y, z) \in \mathbb{E}^3$ is a point in 3D Euclidean space. The element $v_i(x, y, z)$ is a scalar, vector or tensor defined by:

- a scalar field $f : \mathbb{E}^3 \rightarrow \mathbb{R}$

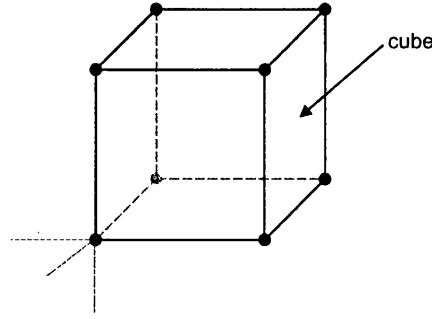


Figure 2.4: Relationship between a voxel and a cell (regular grid)

- an n -dimensional vector function $f^n : \mathbb{E}^3 \rightarrow \mathbb{R}^n$
- a k -ranked tensor function $f^{n^k} : \mathbb{E}^3 \rightarrow \mathbb{R}^{n^k}$

Common volume dataset elements are scalars and vectors which are special cases of tensor functions. These functions are rasterized lookup tables since scanners produce a digitized output of discrete locations on a rectilinear grid. These lookup table functions are held in memory and return an element from a specific location addressed on a rectilinear grid.

Using *f-rep* models allows infinite resolution where a rasterized memory representation will have at best the acquisition methods output resolution. Non-regular grids, unstructured grids and non-regular unstructured grids exhibit cells of different shapes and volumes, adapted rendering techniques must be used to traverse these structures correctly.

The notion of a *voxel* describes the value at an elements location. A voxel (volume element) is analogous to a pixel (picture element) however instead of being a two dimensional element, a voxel is a three dimensional element. When dealing with rectilinear grids a cell is a uniform cube that occupies a unit space and volume defined by the acquisition methods output resolution. The eight corners of a cell represent voxel values on the rectilinear grid (see Figure 2.4).

2.2 Signal Reconstruction Filters

Often the original objects resolution must be synthesised as intricate detail within one sample or voxel and may be missed as a result of the acquisition resolution or rasterization. A reconstruction filter can be used to approximate the original objects continuous signal.

Most volume datasets are rasterized three dimensional blocks of data obtained from scanning equipment. Therefore only a partial mapping from coordinates defined in the volume space exist to voxels. Voxel values defined at the corners of a cell do not necessarily represent a continuous signal inside the cell, minute details smaller than the resolution of the raster resolution are lost. To accurately reconstruct a signal from samples, twice the frequency of the original signal is required (the Nyquist frequency or critical frequency). A sampling frequency at just above the maximum frequency contained in the original signal allows for perfect reconstruction, however additional oversampling is required to use less

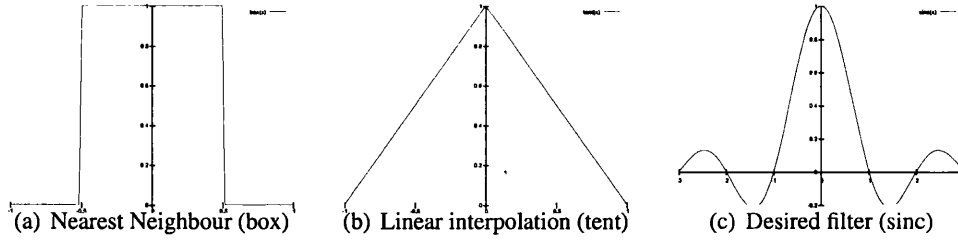


Figure 2.5: 1D Reconstruction filters

complex reconstruction filters (generally twice the frequency of the original). Since acquired volume data will in most cases be scanned at a frequency less than twice the original maximum frequency, a perfect reconstruction can not be obtained from the scanned volume. A close approximation can be obtained using interpolation within reconstruction filters. If the acquired volume data is scanned at twice the maximum frequency of the original signal, a perfect reconstruction can be obtained using the *sinc* filter. Figure 2.5 depicts 1D filters.

2.2.1 Nearest Neighbour

The *nearest neighbour* or *box* reconstruction filter for a point within a cell is the nearest element or voxel value spatially:

$$V(x, y, z) = V(\lfloor x + 0.5 \rfloor, \lfloor y + 0.5 \rfloor, \lfloor z + 0.5 \rfloor) \quad (2.3)$$

Where $(x, y, z) \in \mathbb{E}^3$ is a point in Euclidean space.

This approach delivers a fast and easy implementation, however the image quality is poor due to not attempting to calculate an approximation of the original signal. This results in under-sampling the original signal by simply using the raster blocks original resolution. Figure 2.5(a) shows a 1D nearest neighbour or box filter kernel.

2.2.2 Trilinear Interpolation

Trilinear interpolation attempts to reconstruct the original signal by means of averaging the eight neighbouring elements. A point inside a volume cell can be trilinearly interpolated by firstly linearly interpolating the bottom and top edge of the front face, the same process is then applied on the bottom and top edges of the back face. These two linearly interpolated points are then used for a third linear interpolation between the computed points:

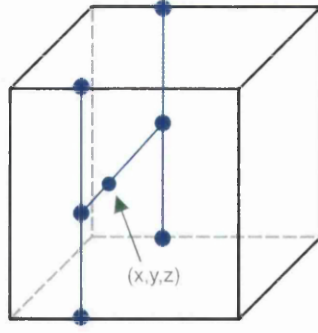


Figure 2.6: Trilinear interpolation in a volume cell

$$\begin{aligned}
 \mathbb{V}(x, y, z) = & \mathbb{V}(\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor) (1 - (x - \lfloor x \rfloor)) (1 - (y - \lfloor y \rfloor)) (1 - (z - \lfloor z \rfloor)) + \\
 & \mathbb{V}(\lceil x \rceil, \lfloor y \rfloor, \lfloor z \rfloor) (x - \lfloor x \rfloor) (1 - (y - \lfloor y \rfloor)) (1 - (z - \lfloor z \rfloor)) + \\
 & \mathbb{V}(\lceil x \rceil, \lceil y \rceil, \lfloor z \rfloor) (x - \lfloor x \rfloor) (1 - (y - \lfloor y \rfloor)) (z - \lfloor z \rfloor) + \\
 & \mathbb{V}(\lfloor x \rfloor, \lfloor y \rfloor, \lceil z \rceil) (1 - (x - \lfloor x \rfloor)) (1 - (y - \lfloor y \rfloor)) (z - \lfloor z \rfloor) + \\
 & \mathbb{V}(\lfloor x \rfloor, \lceil y \rceil, \lfloor z \rfloor) (1 - (x - \lfloor x \rfloor)) (y - \lfloor y \rfloor) (1 - (z - \lfloor z \rfloor)) + \\
 & \mathbb{V}(\lceil x \rceil, \lceil y \rceil, \lfloor z \rfloor) (x - \lfloor x \rfloor) (y - \lfloor y \rfloor) (1 - (z - \lfloor z \rfloor)) + \\
 & \mathbb{V}(\lceil x \rceil, \lceil y \rceil, \lceil z \rceil) (x - \lfloor x \rfloor) (y - \lfloor y \rfloor) (z - \lfloor z \rfloor) + \\
 & \mathbb{V}(\lfloor x \rfloor, \lceil y \rceil, \lceil z \rceil) (1 - (x - \lfloor x \rfloor)) (y - \lfloor y \rfloor) (z - \lfloor z \rfloor)
 \end{aligned} \tag{2.4}$$

Where $(x, y, z) \in \mathbb{E}^3$ is a point in Euclidean space. Figure 2.6 demonstrates trilinear interpolation in a volume cell

This approach is relatively fast and easy to implement. The image quality produced is comparable to higher order interpolation schemes. Trilinear interpolation provides a good trade off between speed and quality. Many modern graphics processors include trilinear interpolation operations implemented in hardware. However software implementations take several operations and could potentially become a bottleneck. Figure 2.5(b) shows a 1D linear interpolation or tent filter kernel.

2.2.3 Tricubic Interpolation

Tricubic interpolation uses a 64 voxel neighbourhood to approximate the original signal. Mitchell and Netravali [MN88] implement a BC family of cubic splines:

$$k_{B,C}(t) = \frac{1}{6} \begin{cases} (12 - 9B - 6C)|t|^3 + (18 + 12B + 6C)|t|^2 + (6 - 2B) & \text{if } |t| < 1 \\ (-B - 6C)|t|^3 + (6B + 30C)|t|^2 + (-12B - 48C)|t| + (8B + 24C) & \text{if } 1 \leq |t| < 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Where B and C are used to control the spline kernel.

To reconstruct a non integer spatial location's intensity:

$$\mathbb{V}(x, y, z) = \sum_{n=-1}^2 \sum_{m=-1}^2 \sum_{l=-1}^2 \left(\begin{array}{c} \mathbb{V}(\lfloor x \rfloor + l, \lfloor y \rfloor + m, \lfloor z \rfloor + n) \\ k_{B,C}(n - (z - \lfloor z \rfloor)) k_{B,C}(m - (y - \lfloor y \rfloor)) \\ k_{B,C}(l - (x - \lfloor x \rfloor)) \end{array} \right)$$

Where $(x, y, z) \in \mathbb{E}^3$ is a point in Euclidean space.

Tricubic interpolation does produce better approximations of the original signal than trilinear interpolation. Most often these differences are marginal in most circumstances, however allow higher quality output for large images. There is also an issue when computing values close to volume edges as a 64 voxel neighbourhood will not be present, lower order interpolation techniques such as trilinear interpolation can be employed near volume edges as a substitute.

The algorithm is more demanding to compute than lower order interpolation techniques and therefore is used for high accuracy rendering that is not heavily dependent on time. There has been several publications [MMMY96, MMY97, MMK⁺98] geared towards higher order filtering specifically for volume visualisation signal reconstruction.

Recent advancements [HE99, HTG01, HVTH02, HHM03] have allowed real time rendering with tricubic filters using modern consumer level hardware. Multiple passes are employed to convolute with filters over 3D volume textures or volume datasets. Figure 2.5(c) shows a 1D cubic filter kernel.

2.3 Volume Data Classification

The fundamental advantage to volume visualisation is that a dataset can represent multiple objects in one structure, additionally internal details of objects are also represented. For example the *CTHead* dataset contains information for bone and skin by encoding the densities discovered at scanned locations. Visualisations can be generated to concentrate on a particular medium within the volume via the process of *segmentation*. Most segmentation tasks are automatic as the whole volume is to be considered for the final image. Certain regions of interest (ROI) within volumes require a specific segmentation that in some cases are not automatic and require a pre-processing step or user interaction.

Classification is the process of assigning a colour and an opacity to a voxel chosen for display, this is done with a transfer function containing colour and opacity information. Transfer functions can either be categorized as *binary* or *fuzzy* segmentation [THB⁺90]. Volumes predominately encode an intensity as a scalar (see Equation 2.2) that will produce grey scales if rendered directly. The opacity property of a transfer function can be used to segment the volume data in an automatic manner and is defined by an *opacity transfer function* (see Eqn 2.6). The colour component of a transfer function is defined similarly in a *colour transfer function* (see Equation 2.7). These two functions are usually combined into a single transfer function and encoded as a rasterized look up table:

$$\alpha : \mathbb{E}^3 \rightarrow [0, 1] \quad (2.6)$$

where $\alpha(x, y, z)$ defines the opacity at $(x, y, z) \in \mathbb{E}^3$, a point in Euclidean space.

$$rgb : \mathbb{E}^3 \rightarrow [0, 1]^3 \quad (2.7)$$

where $rgb(x, y, z)$ defines the colour at $(x, y, z) \in \mathbb{E}^3$, a point in Euclidean space. $[0, 1]^3$ defines a colour triple $\langle r, g, b \rangle$.

To *iso-surface* (see section 2.5 for definition) a volume dataset for a given *iso-value* ϕ , an opacity transfer function is defined. This opacity transfer function will perform a binary segmentation required for the iso-surface:

$$\alpha(x, y, z) = \begin{cases} 1 & \text{if } \mathbb{V}(x, y, z) \geq \phi \\ 0 & \text{otherwise} \end{cases}$$

Volume datasets can also exhibit multiple iso-surfaces (for example the *CTHead* as described above). The opacity transfer function can be encoded to cope with multiple (n) iso-surfaces by encoding different opacity information to each voxels intensity (α_i). The ordering of iso-values within the transfer function is important to describe which surfaces are considered first. This mapping of arbitrary opacity value to arbitrary voxel intensities is fuzzy:

$$\alpha(x, y, z) = \begin{cases} \alpha_1 & \text{if } \mathbb{V}(x, y, z) \geq \phi_1 \\ \vdots & \vdots \\ \alpha_n & \text{if } \mathbb{V}(x, y, z) \geq \phi_n \\ 0 & \text{otherwise} \end{cases}$$

The opacity transfer function for fuzzy classification can be categorized as a one dimensional transfer function as there exists a one-to-one mapping between voxel intensity and opacity value. Higher order opacity transfer functions can be constructed by utilizing further information from neighbouring voxel values. There is not always a perfect interface between two boundaries described in a volume. For example the *CTHead* boundary between skin and bone can exhibit intensities that could belong to both mediums.

Levoy [Lev88] describes an opacity transfer function that calculates a weighting to describe if a voxel is included in the object of interest. The length of the *gradient vector* (see section 2.4) is used to address this weighting by assuming that the gradient will be longer if it lies in a well founded boundary. This transfer function is relatively generic for medical volume data:

$$\alpha(x, y, z) = \begin{cases} 1 & \text{if } |\nabla \mathbb{V}(x, y, z)| = 0 \text{ and } \mathbb{V}(x, y, z) = \tau \\ 1 - \frac{1}{\sigma} \left| \frac{\tau - \mathbb{V}(x, y, z)}{|\nabla \mathbb{V}(x, y, z)|} \right| & \text{if } |\nabla \mathbb{V}(x, y, z)| > 0 \text{ and} \\ & \mathbb{V}(x, y, z) - \sigma |\nabla \mathbb{V}(x, y, z)| \leq \tau \text{ and} \\ & |\nabla \mathbb{V}(x, y, z)| \geq \tau \\ 0 & \text{otherwise} \end{cases} \quad (2.8)$$

Where σ is the width in voxels of the neighbourhood of the iso-value τ .

Algorithms for automatic and semi-automatic generation of transfer functions have been developed [PLB⁺01]. Bajaj *et al.* [BPS97] use a data-centric method for estimating an iso-value to derive a transfer function. He *et al.* [HHKP96] use genetic algorithms to iteratively refine on a transfer function. This process can involve user interaction during the refinement process or can be fully automated.

Kindlamann and Durkin [KD98] employ a two dimensional histogram scatter plot of the volume signal and gradient magnitudes to determine possible boundaries. They present a semi-automatic transfer function generator via user analysis of the scatter plot. Kniss *et al.* [KKH01] extend this work by introducing a set of direct manipulation widgets as an interface for defining multidimensional transfer functions.

Ma *et al.* [Tze05, TLM03] introduce algorithms to automate transfer function generation by directly painting on volume dataset slices to select regions of interest. Machine learning is employed to extract a transfer function for the whole volume dataset based upon the users initial interaction on a small subset of the volume.

The best results for generation of transfer functions is still considered to be achieved best with manual user interaction. This process is aided by providing the user information about the volume dataset in the form of histograms or scatter plots. The goal is to provide an automatic transfer function generator for a volume dataset to remove the burden of transfer function design from the user, allowing novice users powerful volume visualisation tools without having to learn about classification.

2.3.1 Pre-Classification

Pre-classification is a pre-processing step carried out before rendering to transform raw voxel values into colour and opacity quadruples. A reconstruction filter can be applied during rendering to obtain interpolated colour and opacity values. Interpolating between samples that are classified produces a blurring of the data.

Pre-classification is unable to represent high frequency details in the original signal correctly due to signal reconstruction being applied after a colour and opacity have been assigned. Pre-classification is the fastest form of classification, especially when attempting real-time display. Interactive change of the transfer function is not always possible due to the expensive requirement to recompute the classification pre-processing step. Figure 2.7(d) shows a pre-classified image.

2.3.2 Post-Classification

Post-classification occurs after a reconstruction filter has been applied to the sampled signal. Applying reconstruction filters before classification results in the best possible approximation of the original signal. Classification is applied to the approximation of the original signal and therefore accurately accounts for high frequencies within the volume due to signal reconstruction. Post-classification also provides a means for faster interactive transfer

function change because only updating the transfer function is required. Figure 2.7(c) shows a post-classified image after interpolation where high frequencies are maintained.

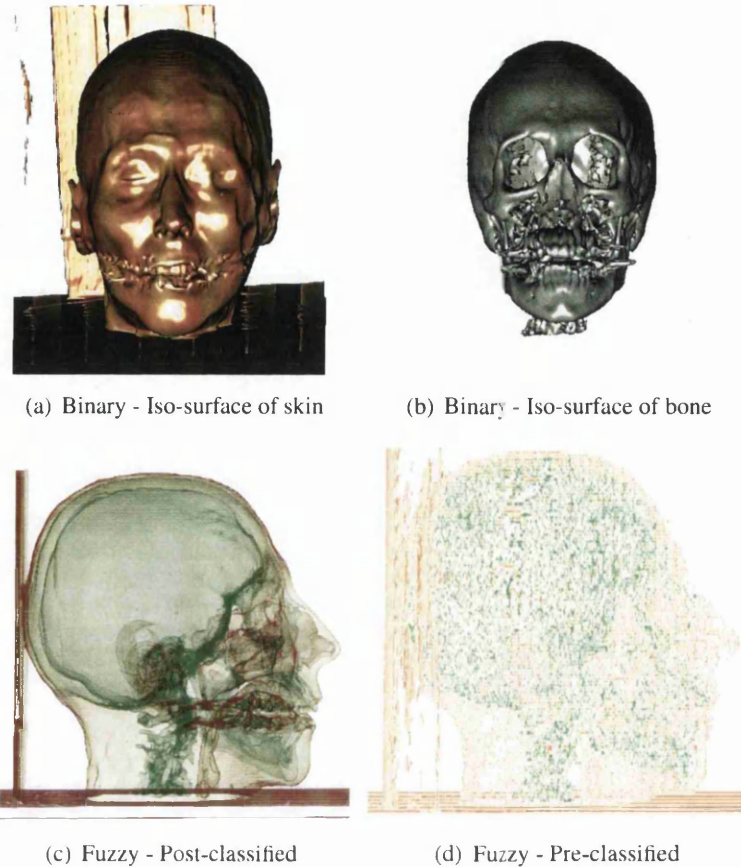


Figure 2.7: Classification examples of the *CTHead* dataset: (a) is binary segmented iso-surface with the chosen density representing skin; (b) is binary segmented iso-surface with the chosen density representing bone; (c) is a fuzzy segmentation applied after signal reconstruction (post-classification) with red depicting skin density and green depicting bone density; (d) is a fuzzy segmentation applied before signal reconstruction (pre-classification) with red depicting skin density and green depicting bone density. Each example is generated using image-order DVR techniques with interpolation applied between samples. These techniques are explored later in this thesis.

2.4 Gradient Computation and Shading

Understanding of visualisations is greatly increased by providing visual cues. Simply providing the colour and opacity information after classification often does not adequately portray spatial detail. A spatial perception of depth and orientation is required to provide an intuitive and easily recognisable image. *Depth cuing* [FvDFH96] introduces a level of depth perception by proportionally encoding the depth reached from the image plane at the closest sample point in the volume into the final image pixel's intensity. Thus contributed depth

intensity decreases further away from the image plane. The depth value to alter the final pixels intensity can be calculated using a *z-buffer*. Depth cuing is fast to compute but does not describe orientation. An extension to this method [THR97] uses gradient differences within the z-buffer to construct gradient normal estimates.

Methods for adding visual depth perception and orientation information to an image involve shading. Gradient normals from the original volume data are computed and used in a lighting model to derive a colour at final image pixels. Section 2.4.1 examines computing gradient normals and section 2.4.2 examines lighting models.

2.4.1 Gradient Computation

Hohne and Bernstein [HB86] use *central differences* to construct a volume datasets gradient normals. The differences in density in a voxels surrounding neighbourhood is used to derive the surrounding gradients of density which are combined to form the gradient normal for that voxel $G : \mathbb{E}^3 \rightarrow \mathbb{R}^3$. The simplest central difference method considers the 6 face connected surrounding voxels:

$$\begin{aligned} G_x &= \mathbb{V}(x+1, y, z) - \mathbb{V}(x-1, y, z) \\ G_y &= \mathbb{V}(x, y+1, z) - \mathbb{V}(x, y-1, z) \\ G_z &= \mathbb{V}(x, y, z+1) - \mathbb{V}(x, y, z-1) \end{aligned} \quad (2.9)$$

Where $(x, y, z) \in \mathbb{E}^3$ is a point in Euclidean space and is a location on the rectilinear grid.

Gradients with increased accuracy can be computed by considering 26 neighbours:

$$\begin{aligned} G_x &= \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x+1, y+\alpha, z+\beta) - (x-1, y+\alpha, z+\beta) \\ G_y &= \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x+\alpha, y+1, z+\beta) - (x+\alpha, y-1, z+\beta) \\ G_z &= \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x+\alpha, y+\beta, z+1) - (x+\alpha, y+\beta, z-1) \end{aligned} \quad (2.10)$$

If neighbouring voxels on a plane are both higher or lower than the voxel being computed forward or backward differences are used to replace central differences.

26 connected forward differences:

$$\begin{aligned} G_x &= \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x+1, y+\alpha, z+\beta) - (x, y+\alpha, z+\beta) \\ G_y &= \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x+\alpha, y+1, z+\beta) - (x+\alpha, y, z+\beta) \end{aligned}$$

$$G_z = \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x + \alpha, y + \beta, z + 1) - (x + \alpha, y + \beta, z)$$

26 connected backward differences:

$$G_z = \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x + \alpha, y + \beta, z) - (x + \alpha, y + \beta, z - 1)$$

$$G_x = \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x, y + \alpha, z + \beta) - (x - 1, y + \alpha, z + \beta)$$

$$G_y = \sum_{\beta=-1}^1 \sum_{\alpha=-1}^1 \mathbb{V}(x + \alpha, y, z + \beta) - (x + \alpha, y - 1, z + \beta)$$

This 26 connected method will not correctly process a volume dataset's borders, so 6 connected central differences are computed in these situations.

After calculating the gradients they are normalised to $N = (n_x, n_y, n_z)$

$$n_i = \frac{G_i}{\sqrt{G_x^2 + G_y^2 + G_z^2}} \quad (2.11)$$

Additionally a filtering step can be employed to smooth the gradient normal vector field. This step can improve visual aesthetics because discontinuities in the gradient field produce sharp edges in the final output.

2.4.2 Shading

Gradient shading algorithms that consider depth and surface orientation provide the best visual perception. These algorithms require a gradient normal which is analogous to a normal vector in surface graphics, which represents a normalized vector pointing away from a surface. Since volume techniques can represent substrates with no surface the term gradient normal is used in favour of surface normal since it is derived from the rate of change from surrounding voxels. Grey level shading [HB86] uses gradient normals from grey level data to incorporate directional shading. This method has largely been adapted to work with lighting and colour classification.

Several illumination models are defined in surface graphics that model light contribution, scattering and surface properties such as colour, reflectance and opacity. These models can be defined for volume rendering techniques, however must additionally define appropriate opacity information to be evaluated by the volume rendering light model. Generally ambient light defines the contribution of random low level light throughout a scene, diffuse light models the contribution light emitted from an object and specular light defines light that is reflected from an object.

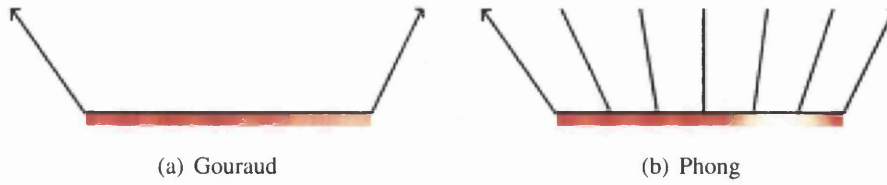


Figure 2.8: Normal vectors defined for Gouraud and Phong shading models across a primitive face between two vertices. Gouraud shading interpolates the colours defined at each vertex along a primitives face where phong shading firstly interpolates the normal vector across a primitives face and allows more precise illumination.

Phong [Pho75] introduces the *Phong reflection model* which describes the calculation of lighting at each sample point for a set of light sources (see Eqn 2.12). This model is generally regarded as a simplification of the physically based modelling of light transport through a scene in surface graphics.

$$I = I_d k_a C_a + C_p \sum_l k_d I_d (\bar{N} \bullet \bar{L}) k_s I_s (\bar{R} \bullet \bar{V})^n \quad (2.12)$$

where I_d is the intensity of light at a given point, k_a is the ambient reflection co-efficient at a given point, C_a is the ambient light colour, C_p is the light source colour, k_d is the diffuse reflection co-efficient, k_s is the specular reflection co-efficient, I_s is the specular colour, \bar{N} is the normal vector, \bar{L} is the light direction vector, \bar{R} is the reflectance vector, \bar{V} is the view direction vector and l is the number of lights in a light set.

Gouraud [Gou71] introduced per vertex lighting, an approach that calculates a lighting model at each vertex in a polygonal mesh and interpolates the final colour at final image pixels over the surface of a primitive. Normal vectors are calculated for each vertex only and thus do not accurately allow representation of specular contributions. Additionally the resolution of the polygonal mesh determines the quality of the final image since a dense set of primitives allow more accurate results. Gouraud shading is considered a faster alternative to the later more accurate Phong shading model. Figure 2.8(a) details the normal vectors encountered in Gouraud shading.

Phong [Pho75] describes an extension to Gouraud shading where the lighting calculations are applied at each final image pixel. This is also referred to as *per-pixel lighting*. At each vertex, the normal vector is interpolated across the face of each primitive, and the lighting calculations occur after rasterization. This allows greater accuracy in determining the exact contribution of lighting properties at each final image pixel and allows accurate modelling of specular highlights. Figure 2.8(b) details the normal vectors encountered in Phong shading.

Blinn [Bli77] developed an approximation of the original Phong reflectance model. The reflectance vector is computationally expensive to derive and therefore a half angle vector is used to replace this expensive operation. Blinn-Phong shading describes Blinn's approximate reflection model coupled with per-pixel lighting. Although Blinn's approximate reflectance model does not produce the high accuracy of full Phong reflectance with respect to

specular contributions, a suitable approximation allows rendered images to exhibit specular highlights that appear accurate to the viewer.

Kautz [Kau04] provides a thorough review of lighting techniques focusing on hardware implementation. These lighting models can be directly employed in surface graphics and adapted to volume visualisation.

Phong shading is generally used in volume rendering applications where lighting contributions and object surface properties as each sample being considered contains its own gradient normal information. However some object-order techniques can employ the cheaper Gouraud alternative with less visually pleasing results.

2.5 Iso-Surface Reconstruction

Surface tiling or surface tracking algorithms describe a method of obtaining a polygonal mesh from a volume dataset that represents an iso-surface of a given iso-value. An iso-surface is a set of points inside the volume that match a given iso-value:

$$iso(V, \tau) = \{(x, y, z) \in \mathbb{E}^3 | V(x, y, z) = \tau\} \quad (2.13)$$

Where τ is the iso-value or threshold.

This polygonal mesh can be rendered using traditional surface graphics approaches. These methods typically consider gradient normal estimation, or calculate surface normals to include in shading algorithms. This intermediate polygonal representation is capable of producing high quality images for a chosen iso-surface, however does not contain any information describing the interior or exterior of an extracted object.

There are two basic approaches iso-surface reconstruction.

- Surface Tiling - March through the dataset one cell at a time and construct a polygon mesh within each cell of the approximated iso-surface.
- Surface tracking - Start with a given cell that contains the iso-value and construct an iso-surface by visiting neighbouring cells recursively, building a polygonal mesh.

Figure 2.12 shows images obtained by using the marching tetrahedra algorithm on the AVSHydrogen volume dataset ¹.

2.5.1 Contour Tracking

Keppel outlined *contour tracking* [Kep75] as the first method of extracting an iso-surface from a volume dataset. Each 2D planar slice is assessed with an iso-value to construct a set of vertices on the iso-surface. These vertices are connected to form an *iso-contour*. After processing the stacks of planar slices in the volume, the algorithm connects points from each planar iso-contour to form a polygonal mesh representing the iso-surface. Fuchs *et*

¹Thanks to M. W. Jones for marching tetrahedra code

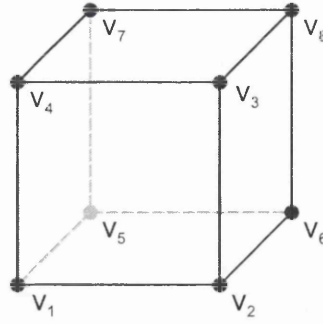


Figure 2.9: Marching cubes vertex enumeration scheme

al. [FKU77] and later Christiansen and Sederberg [CS78] used heuristics to minimise the surface area of triangles produced. These approaches exhibit ambiguity when an iso-surface is not closed, or there are multiple iso-contours. The resolution of the extracted polygonal mesh is also lower than later techniques [JC94].

2.5.2 Marching Cubes

Wyvill *et al.* [WMW86] use surface tiling techniques to extract polygonal iso-surfaces from field functions with an iso-value. The main focus of this work is to enable soft-objects to be represented in surface graphics systems. Known key points describing the iso-surface from the field function in 3D space are stored in a hash table and traversed to find seed cells. Field functions describing stochastic characteristics are explored rather than rasterized volume datasets. Six connected neighbouring cells are then examined from seed cells to determine a set of cells that intersect the iso-surface. These intersected cubes are then tiled with respect to the field function along each cube edge.

Lorenson and Cline [LC87] describe the marching cubes algorithm which visits all cells in a volume and categorises each corner voxel (cell vertex) as being inside or outside the iso-surface. The enumeration scheme for each vertex is given in Figure 2.9. A cell is considered *transverse* if the iso-surface is discovered to be inside this cell. This categorization is achieved by constructing an eight bit word encoding each vertices respective state; inside the boundary (1) or outside the boundary (0).

An eight bit word $b = b_8b_7b_6b_5b_4b_3b_2b_1$ as an unsigned integer representing each vertices respective binary state:

$$b_i = \begin{cases} 1 & \text{if } \mathbb{V}(v_i) \geq \tau \\ 0 & \text{otherwise} \end{cases} \quad (2.14)$$

Where i corresponds to a bit in the unsigned eight bit integer, $v_i \in \mathbb{E}^3$ is a point in Euclidean space and τ is the iso-value.

A cell is transverse if the constructed eight bit integer satisfies:

$$0 < b < 255$$

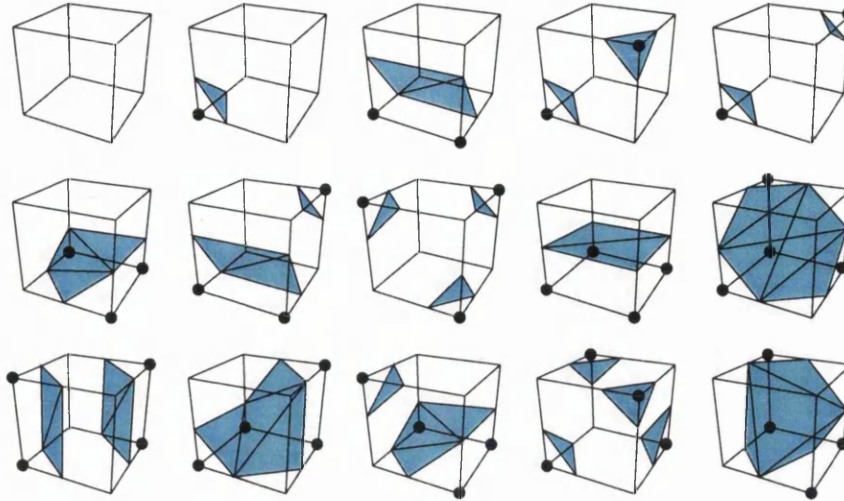


Figure 2.10: Marching cubes basic case table

A transverse cell is tiled according to a case lookup table defined with the eight bit integer. There are 14 base cases in the marching cubes algorithm although there are 256 cases in total ($2^8 = 256$ cases based on 8 vertices's). The total cases can be simplified to 14 cases (Figure 2.10) because many cases are derivable under symmetry, rotation and inversion. These simplifications can leave ambiguities in certain cases and are not topologically equivalent to the full case table. Interpolation is used in each transverse cell to position vertices in the generated mesh. The linear interpolation function approximates each vertex being positioned on the iso-surface along a given cell edge. This ensures that vertices along edges of a cell correspond to neighbouring sub meshes:

The interpolation function:

$$\gamma = v_i + \frac{\tau - \mathbb{V}(v_i)}{\mathbb{V}(v_j) - \mathbb{V}(v_i)}(v_j - v_i) \quad (2.15)$$

Where τ is the iso-value, $v_i, v_j \in \mathbb{E}^3$ are points in Euclidean space and represent vertex locations at each end of the edge being considered. γ is the interpolated value along the edge being considered.

Dürst [Dür88] noted that due to simplification, the original algorithm suffers from ambiguity problems. These ambiguities occur when two opposing vertices are diagonally opposite within a cell face [WG90, NH91], this leaves holes in the final mesh. This is due to using linear interpolation along a cells edge, and an over simplified case table.

Neilson and Hamann [NH91] solved these ambiguous case problems by using bilinear interpolation across a cells face and choosing a correct triangulation based upon an asymptotic decider. Vertices are also added in the cell rather than strictly along edges. They further adapted existing case tables for all possible ambiguous cases in the original algorithm. This solves ambiguity issues for the bilinearly interpolated faces and is topologically correct.

Natarajan [Nat94] and Chernyaev [Che95] recognised ambiguities that arise inside a cell in certain cases. They interpolate the internal body with trilinear interpolation and decide

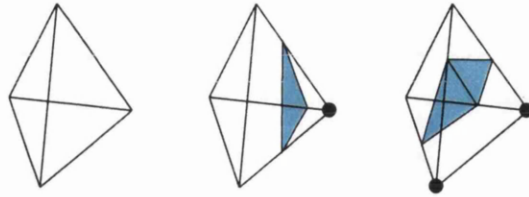


Figure 2.11: Marching tetrahedra basic case table

upon an appropriate extension case. These methods are topologically correct. Hamman *et al.* [HTF97] employ quadratic bezier patches to improve the accuracy of iso-surface reconstruction within a cell.

Lopes and Brodlie [LB03] refine upon internal cell ambiguity and accuracy by considering points within a cell which are important to correctly represent the iso-surface. Neilson [Nie03] further detailed and refined the above algorithms to include trilinear interpolation within the cell and a three level case table. This method solves ambiguous cases within a cell and produces non-ambiguous meshes.

2.5.3 Marching Tetrahedra

The marching tetrahedra algorithm is similar to marching cubes, however uses tetrahedra as the base primitive for triangle consideration. Each cell is subdivided into tetrahedra. Bloomenthal [Blo88] splits a cell into six, five cornered pyramids and each pyramid is subdivided into two tetrahedra. This work is used on implicit surfaces to construct iso-surfaces. Payne and Toga [PT90] and later Neilson and Sung [NS97] recognised the ambiguity problems in the original marching cubes implementation. They resolved these ambiguities by simplifying a cell by subdividing it into five tetrahedra. Each tetrahedra within a cell can be tiled with zero, one or two triangles depending the iso-surfaces intersection. These three basic cases (see Figure 2.11) are simplified from the sixteen possible cases ($2^4 = 16$ cases based on 4 vertices) by exploiting symmetry, rotation and inversion. See Figure 2.12 for example renderings with the marching tetrahedra algorithm.

Triangle meshes produced using the marching tetrahedra are topologically simple and correct, iso-surfaces are also correctly closed due to removing ambiguity problems. Whilst the algorithm is simple, the resulting triangular mesh complexity is much greater than meshes produced from the marching cubes. A post-processing step can be introduced [HDD⁺93, SZL92, Tur92, ILGS03] to reduce mesh complexity, however this can lead to under-sampling the iso-surface due to multiple triangles being over simplified.

2.5.4 Iso-Surface Tracking

Surface tracking algorithms [WMW86, ZJ91] operate by following the surface from one location outward recursively. Surface tracking algorithms do not require a pre-processing step to determine if a cell is transverse. A transverse seed cell must either be discovered or

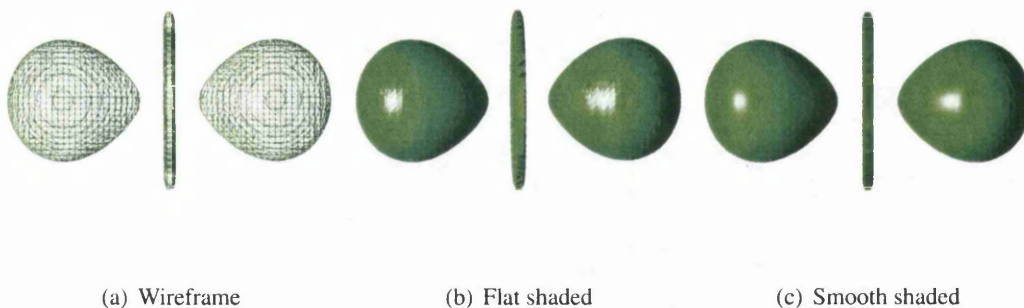


Figure 2.12: AVSHydrogen dataset images from the marching tetrahedra algorithm

known to begin tracking. A cell that contains the surface is tiled and neighbouring cells are evaluated to ascertain if they also contain the iso-surface.

Generally a 14 connected strategy is used; cells that are vertex, edge or face connected are considered for surface reconstruction. If the iso-surface is discovered in neighbouring cells, these cells are recursively processed, cells discovered during this recursion are added to a stack to be processed later. Processing continues until an iso-surfaced cell is discovered, at this point the seed cell becomes the top of the stack.

The result is a polygonal mesh describing the iso-surface. These algorithms exhibit a reduction in complexity due to considering less cells but only reconstruct closed surfaces correctly. This method is not well defined if there are multiple surfaces in the volume dataset sharing the same iso-value. Without a known seed point, the algorithm must scan the volume to discover a starting cell. This can be expensive in some cases, although usually starting from the centre of a face and working inward towards the centre of the volume will discover the iso-surface with few lookups.

2.5.5 Octree Encoding

A surface tiling algorithm must visit every cell within the volume in order to decide if the iso-surface is present. Wilhelms and Gelder [WG92] pre-process the volume dataset to learn as much information as possible, the resulting pre-processed data structure is a min-max *octree* [Mea82]. They state that the time spent in surface reconstruction visiting empty cells is between 30% and 70%.

An octree is a higher order binary tree where the root node represents the whole volume, and subsequent branches represent one of eight sub cubes. The minimum and maximum intensity for each sub cube is recorded at each branch for discovering empty regions. Other information can also be stored, for example average values for level of detail rendering. The octree's leaf nodes are the original voxel data. Traversing this structure during surface reconstruction facilitates skipping empty regions of the volume for the iso-value being considered. Figure 2.13 shows an octree structure.

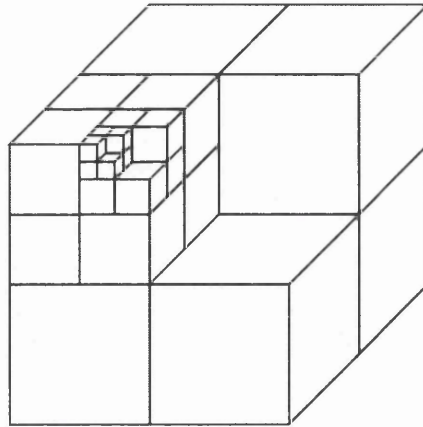


Figure 2.13: Octree data structure overview

2.6 Direct Volume Rendering

Direct volume rendering (DVR) methods produce a visualisation from raw volume data. There is no intermediate surface representation as described in section 2.5. Surface representations can extract iso-surfaces from volume data, however boundaries between differing mediums in the volume dataset are difficult to represent due to binary segmentation at best. Surface techniques also do not allow any representation of amorphous or gaseous phenomena intrinsically. Multiple iso-surface extraction in surface representations include problems due to semi-transparency and self occluding surfaces.

DVR methods can utilize binary or fuzzy classification enabling a higher-order of objects to be visualized. Surface based techniques grow with increasing object complexity, where the complexity of DVR methods remains constant for increasing object complexity since an object has usually been rasterized into a volume. Implicit surfaces and f-rep models are sampled at regular intervals if not rasterized as a pre-processing step, thus rendering complexity is also constant for these models. DVR methods are well defined for representing multiple surfaces, semi-transparent medium, amorphous phenomena, internal and external object detail. Direct volume rendering algorithms can be categorised into image-order (backward mapping) and object-order (forward mapping) approaches. Algorithms can also exhibit characteristics from both approaches, these are hybrid algorithms.

- *Image-Order* algorithms concern deciding which voxels are visible from a particular image position in the image plane.
- *Object-Order* algorithms concern processing the volume and deciding which voxels will be projected onto the image plane.
- *Hybrid approaches* combine features in image-order and object-order approaches.

A DVR algorithm attempts to approximate the *volume rendering integral* [KH84, Max95, Bli82] which describes the passage of light through a volume dataset or volumetric scene. There are many differing light transport models for volumes. The most common model described in equation 2.16 is the *absorption-emission* model [Max95]. Absorption models

light interacting with a density encountered in the volume and can both extinguish or scatter. Emission models light that passes through the volume after considering absorption. A uniform white light in the background of the volume is assumed. Particles are modelled to absorb and emit light, a pixels final intensity I of wavelength λ is derived from discrete samples along a ray inside the volume. The volume rendering integral in its low-albedo form:

$$I_\lambda = \int_0^L C_\lambda(s) \mu(s) e^{-\int_0^s \mu(t) dt} ds \quad (2.16)$$

where light rays of length L composite light $C_\lambda(s)$ of wavelength λ reflected at point s in the direction ds . These wavelengths usually consist of red, green and blue for an RGB colour space. Other colour spaces can be used [ARC05, Max95], however most computer monitors and graphics hardware adopt the RGB model. A mapping would be required for other colour models.

For computation purposes this model must be simplified to allow discretised rays to approximate the lighting equation though composited discrete samples. The Riemann sum can be used to approximate and discretise the continuous volume rendering integral:

$$I_\lambda = \sum_{i=0}^n C_\lambda(i\Delta s) \mu(i\Delta s) \Delta s \prod_{j=0}^{i-1} \exp(-\mu(j\Delta s) \Delta s)$$

where n is the number of samples taken at separation distance $\Delta s = \frac{L}{n}$ along the ray.

A further approximation can be achieved using the first two terms of the Taylor expansion for the exponential term:

$$I_\lambda = \sum_{i=0}^n C_\lambda(i\Delta s) \alpha(i\Delta s) \prod_{j=0}^{i-1} (1 - \alpha(j\Delta s))$$

During DVR the volume rendering integral is usually calculated using the following compositing equations. This compositing equation is the discretised per sample integration step. Back-to-front order compositing and front-to-back order compositing are performed differently. Additionally associated colours (or opacity weighted colours) can be used where the colour is pre-multiplied with its corresponding opacity. Using associated colours can reduce artifacts in the final rendering [WMG98, Bli94].

Porter and Duff [PD84] define the *over* operator which is the per-sample integration step defined in these operations. They describe further additional compositing operations for more specific applications. Equation 2.17 details associated colouring, equations 2.18 and 2.19 describe back-to-front, and front-to-back compositing using the *over* operator for each sample.

$$c' = c\alpha \quad (2.17)$$

where $c' \in [0, 1]^3$ is an $\langle r, g, b \rangle$ triple and $\alpha \in [0, 1]$ the associated colour is usually stored with the alpha component in an $\langle r, g, b, \alpha \rangle$ analogous to a non associated colour.

$$c'_i = c_i + (1 - \alpha_i)c'_{i+1} \quad (2.18)$$

where c'_i is the composited colour value for sample i and $c'_i, c_i, c'_{i+1} \in [0, 1]^3$ are $\langle r, g, b \rangle$ triples and α_i represents the opacity component of the $\langle r, g, b, \alpha \rangle$ quadruple. The alpha component of previously composited colours is not required with associated back-to-front compositing.

$$c'_i = c'_{i-1} + (1 - \alpha'_{i-1})c_i \quad (2.19)$$

$$\alpha'_i = \alpha'_{i-1}(1 - \alpha'_{i-1}) + \alpha_i$$

where c'_i is the composited colour value for sample i and α'_i is the composited opacity value for sample i , $c'_i, c_i, c'_{i-1} \in [0, 1]^3$ are $\langle r, g, b \rangle$ triples and α_i represents the opacity component of the $\langle r, g, b, \alpha \rangle$ quadruple. The alpha component of previously composited colours is required for front-to-back compositing.

The number of samples along a ray effects the over operator, more samples generate a more opaque image. Alpha correction can be employed to weight the compositing with respect to the number of samples taken along the ray and produce consistent images for differing quantities of samples.

Alpha Correction:

$$\alpha' = 1 - (1 - \alpha)^{\frac{\Delta t}{\Delta s}} \quad (2.20)$$

where α' is the new alpha obtained from the original alpha value α , Δt is the sampling distance and Δs is the original sampling distance for the transfer function which is usually 1.

Another important compositing operator is the maximum intensity projection (MIP) operator [ASK94]. MIP does not composite along a ray, instead the largest sample value encountered along the ray is used for final pixel intensity. A comparison of the largest voxel value found so far at each discrete ray sample is used.

When processing a discretised ray at regular sampling intervals:

$$d_{i+1} = \max(d_i, d_s); \quad (2.21)$$

where $d_i \in \mathbb{R}^3$ is the maximum encountered scalar along the ray so far and $d_s \in \mathbb{R}^3$ is the scalar for the current sampling position along the ray.

MIP visualisations provide extremely useful visual cues to a user in an intuitive manner (see Figure 2.14). This method is often most useful in medical visualisation where the images are similar to traditional x-rays, although x-ray absorption is not modelled and the information presented is different than a x-ray which displays densities along a 2D plane through an object instead of the maximum densities encountered through a 3D object. It is

possible to model pseudo x-ray absorption in a 3D object with the absorption only volume rendering integral during DVR evaluation of a volume dataset [Max95].

MIP algorithms must consider all of the samples along the ray in order to render an image. This is due to requiring the maximum value along a ray. Thus this methods complexity defines an upper bound or worst case for DVR algorithms as it must process or consider every sample point during discrete ray traversal. The general run-time of computing each ray step is drastically reduced because no compositing operation needs to be performed and the *max* function is extremely efficient. There are also no gradient normals or complex shading functions to compute.

It is also possible to normalise the intensity that is rendered by the *min* and *max* values of the volume dataset. This will yield a greater resolution of intensity when the range of values in the volume dataset are less than the range of available values. MIP algorithms under rotation are not well defined, a rendering of a given viewpoint *a* will be exactly the same as rendering *b* that is 180° around an axis of rotation away from the original viewpoint.

Depth cueing (see section 2.4) techniques can be used to add visual cues for depth perception by assigning the position where the maximum valued scalar was encountered along the ray. Heidrich *et al.* [HMS95] describe a two pass method to add depth cueing to MIP, however this can be achieved in one pass by storing the position where the maximum scalar was encountered along the ray. Rotation is well defined with MIP using depth cueing as renderings 180° around an axis from each other will have differing depth information.

Mora and Ebert [ME05] present an investigation into MIP rendering with tree structures and detail an optimised algorithm that uses hierarchical occlusion maps in an object based manner. They further investigate the complexity and derive an optimised traversal algorithm for image and object based approaches. This provides a lower bound to MIP rendering in average cases by eliminating traversal of several octree nodes. The previous complexity of $O(n^3)$ is simplified to $O(n^2)$ in average cases. The worst case still demonstrates $O(n^3)$ complexity.

2.6.1 Image-Order Algorithm

Image-order approaches [Sab88, UK88, DCH88] originally developed by Levoy [Lev88] have been the main focus of development for volume visualisation. These trends had been noticed earlier by Kajiya *et al.* [KH84]. These methods allow direct computation of a visualisation from the raw volume data. No intermediate stages are generated which differs from surface reconstruction (see section 2.5) methods.

The ray casting model essentially fires rays from image pixel from the image plane into the volume, integrating the contribution of each value encountered along the ray. Figure 2.15 provides an operation overview of the algorithm and Figure 2.16 depicts the equidistant sampling points along a discrete ray. Figure 2.17 shows a 2D plane of where possible sample positions could occur on a rectilinear grid and show how an interpolation scheme would be required to attempt recovery of the original signal. Levoy's [Lev88] original algorithm calculated ray entry and exit points in the volume and traversed each ray from back-to-front.

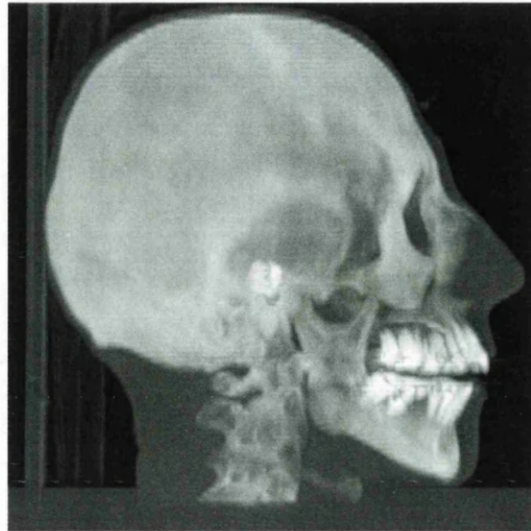


Figure 2.14: MIP rendering of the *CTHead*

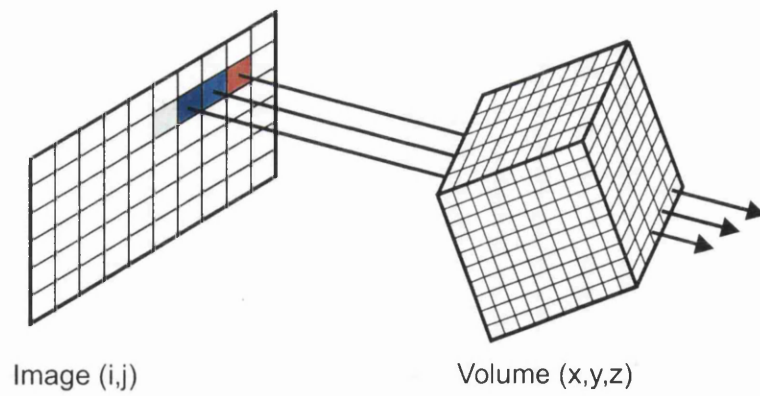


Figure 2.15: Ray-casting overview

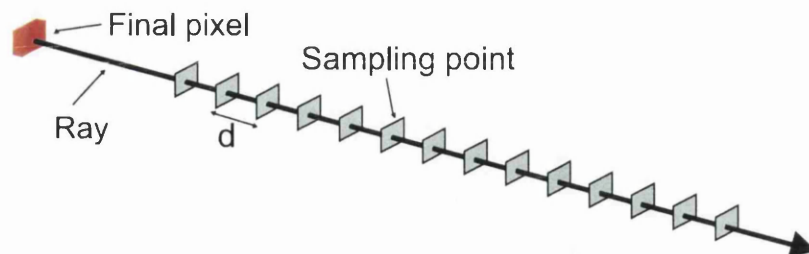


Figure 2.16: Equidistant ray sampling during ray-casting

Samples are generally taken at equidistant points along the ray, Kajiya *et al.* [KH84], Levoy [Lev88] and Sabella [Sab88] all conclude that samples should be taken at regular intervals along the ray when approximating the volume rendering integral. Increasing the number of sample positions along the ray will yield better images, however this comes at the expense of having to calculate a greater number of sampling positions. Many volume datasets that exhibit small structures can be under-sampled by choosing a large stepping distance along the ray. The small structure could be present between the sampling positions and therefore be missed altogether, or introduce an artifact by effecting the interpolation function. Large structures can also be under-sampled producing artifacts in the final image due to the structure not being correctly reconstructed between sample points. Generally the under-sampling of large objects still leaves the end-user with visual cues to the objects general form.

This integration typically follows a volume rendering integral model as discussed earlier in this section. The contributions along the ray are usually the result of a signal reconstruction filter and classification.

Image-order approaches yield the best image results due to a correspondence to sampling at each pixel, this can result in *super-sampling* since multiple rays can be cast for the size of one volume cell. Image-order approaches can also be parallelised since each ray's traversal is disjoint from one another.

Levoy [Lev90] later reported that rendering in a front-to-back order enables the possibility to include *adaptive ray termination*. This observation is based upon an opaque sample occluding any samples further along the ray. Traversing the ray in a back-to-front manner requires the whole ray to be sampled since an opaque sample can occlude samples towards the back of the ray, resulting in unnecessary processing of samples. Traversing the ray in a front-to-back manner allows discovery of opaque samples before processing the remaining samples along the ray. Levoy also reported using an octree structure (see section 2.5.5) to achieve *empty space leaping* by skipping regions of the volume dataset that are empty.

Yagel and Kaufman [YK92] introduce *template based volume viewing* which avoids the costly trilinear interpolation signal reconstruction in image-order rendering. They compute a template of voxels encompassing all voxels to be considered for ray contribution for a particular viewing orientation. This template is moved through the volume to compile the image plane. The template is chosen such that holes in sampling are avoided. Normal ray casting methods can miss voxels based upon step size. The voxels in each template are sampled without any interpolation to derive each sample point. These samples are then

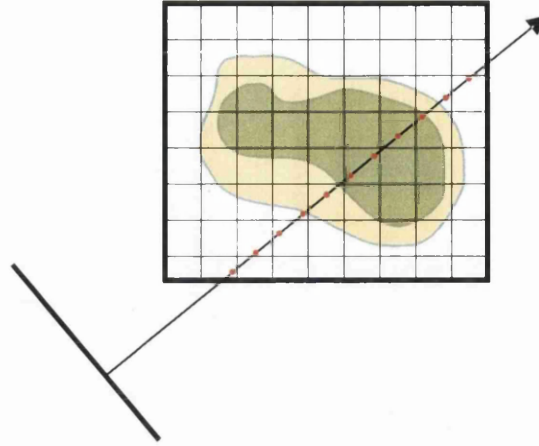


Figure 2.17: 2D ray-casting overview

composed in the standard manner. This algorithm is faster in comparison to standard ray casting techniques because trilinear interpolation is not computed. Images computed with this technique lack the image quality of the original image-order approach using trilinear interpolation.

2.6.2 Object-Order Algorithm

Westover [Wes90] introduced *splatting* as a technique to project voxel contributions to the final image as a footprint. It is possible that a voxels projected contribution to the final image will affect more than one pixel. Westover observes that a voxels energy contribution will fall off from its centre to surrounding mediums. The chosen footprint is spherical to model this behaviour. Since splatting considers voxels independently without grouping voxels together with a ray, each voxel can be processed separately. This allows massive parallel processing of a volume dataset.

This fall off is modelled by a reconstruction filter from a voxel's centre $h : \mathbb{E}^3 \rightarrow \mathbb{R}$. The contribution of a voxel with centre $(i, j, k) \in \mathbb{E}^3$ on a point in 3D Euclidean space $(x, y, z) \in \mathbb{E}^3$ is:

$$\text{contribution}(x, y, z) = \mathbb{V}(i, j, k) h(x - i, y - j, z - k) \quad (2.22)$$

The contribution for a pixel in the final image is the integral of sampled contributions along a line perpendicular to the image plane.

$$\text{contribution}(x, y) = \mathbb{V}(i, j, k) \int_{-\infty}^{\infty} h(x - i, y - j, w) dw \quad (2.23)$$

The original splatting algorithm exhibited some artifacts such as colour bleeding and fuzzy edges. It also is only suitable for rectilinearly defined datasets. Zwicker *et al.* [ZPvBG01,

ZPvG02] introduce elliptical weighted average (EWA) splatting based on elliptical Gaussian kernels. They provide methods to filter reconstruction kernels which reduce aliasing and use different shaped elliptical kernels for the splatting footprint.

The EWA spatting algorithm also works on unstructured and irregular grids and overcomes artifact problems of previous methods. EWA splatting produces high quality splatted images in volume rendering but can also be applied to surface rendering of point clouds. Chen *et al.* [CRZP04] later implemented the EWA splatting algorithm in hardware for interactive display.

Lacroute and Levoy [LL94] report on *shear-warp* rendering, a method to traverse a volume dataset without trilinear interpolation. The premise of the algorithm is to take advantage of spatial coherence between a volume's face plane that is parallel to the final image plane. When the image plane is parallel to a volume's face plane, sampling can occur at regular intervals at each grid location. This approach needs no interpolation as each grid position can be addressed directly, generally bilinear interpolation is used to improve quality within 2D virtual slices.

The shear-warp method traverses a volume dataset by shearing virtual 2D slices that are the most parallel to the image plane, resulting in a stack of sheared slices parallel to the image plane. Conventional object-order rendering is then used to produce an intermediate image from regularly sampled dataset positions. This image is not correct due to the initial shear, a warp operation is then applied to the image to form the correct result. This describes an orthographic projection. A perspective projection is possible by additionally scaling each slice away from the viewpoint.

A run length encoding scheme is used to enable space leaping, each virtual 2D slices z direction is run length encoded as a pre-processing step. This allows fast traversal and efficient memory alignment, although burdens memory consumed. Shear warp is considered the fastest software approach to date, however does exhibit artifacts at 45° rotation angles resulting in a popping effect under rotation. A pre-classification scheme is used to speed up signal reconstruction so fuzzy edges result and this method is prone to under-sampling.

The shear warp algorithm has been implemented in specialist volume rendering hardware. The VolumePRO [PHK⁺99] series of graphics cards use shear warp with trilinear interpolation and super-sampling schemes to take advantage of spatial coherence and memory alignment inherent in the original algorithm. The VolumePRO graphics processor represents the fastest method available to render volumes on a single machine currently.

Both Cabral *et al.* [CCF94] and Cullip and Neumann [CN94] implemented an object-order approach using conventional graphics processing units (GPU). They represent the algorithm by using 3D texture mapping hardware on high end workstations. A set of view aligned slices are rasterized in hardware and subsequently textured from a 3D texture map (or volume dataset) to encode each sampling position's intensity. These are then passed through a fixed function blending operation which provides compositing. Hardware acceleration techniques will be explored further in the next chapter.

2.6.3 Hybrid Approaches

Mora *et al.* [MJC02] introduce *object-order ray casting* as a technique to combine the good features of image-order and object-order approaches. The premise of the algorithm is to locally compute the contribution to the final image for each cell. A hexagonal primitive is used to represent how a cell will map to the image plane when projected. Local ray intersections are pre-computed to approximate image-order ray casting cell traversal. The cell is then sampled with each ray that will pass through it and the appropriate pixels are updated. Each pixel update performs the compositing step along each virtual ray. This method can therefore benefit from leaping empty space in the volume with a min-max octree.

Hong *et al.* [HQB05] implement the object-order ray casting method on consumer level graphics hardware (GPU's) for large datasets. They decompose a dataset into a min-max octree and load non-empty cells into video memory for ray casting. Each cell's traversal is then composited into a final image. The non-empty cells are sorted into planes for compositing order and uploaded to video memory when required.

2.7 Volume Graphics

Volume graphics is a field of computer graphics in its own right that has grown from volume visualisation DVR techniques to provide a superset of graphical methods to the graphics community. Using voxels as a primitive opposed to triangles in surface graphics allows many more features, operations and techniques to be modelled effectively. Kaufman *et al.* [KCY93] examined the emergence of volume graphics as a field in its own right by observing trends in volume visualisation.

Surface based graphics exhibit fundamental problems:

- Surfaces can not accurately represent internal object detail or surface/boundary thickness.
- Surface representations can not represent mediums that are not intrinsically solid. Such mediums include gasses, clouds and particles (amorphous phenomena).
- Semi-transparent objects are difficult to model when there are several meshes overlapping.
- Computational complexity grows with increasing mesh complexity.

Volume based graphics provide intuitive and comprehensive representations:

- Volumes can model internal object detail, a chosen object of interest, external object detail and multiple objects.
- Volumes can model amorphous phenomenon such as gasses, clouds and particles without extending existing techniques.
- Boundaries and surface thickness are well defined.
- Semi-transparent objects and substrates are well defined.

- Computational complexity remains constant for rendering different objects of the same size and additional acceleration methods are possible from the base complexity.
- Volume techniques can import surface and *f-rep* representations as rasterized scalar fields or distance fields.

To represent a graphical model of a human head, (for example the *CTHead*) many thousands of triangles are required in a tessellated mesh, where as a standard volume dataset will be required for volume approaches. Volumetric approaches also yield superior image quality in this case due to optical models being evaluated on rasterized per sample primitives and not defined at each vertex and interpolated across a polygons face.

Having voxels as a primitive object also allows deformation to be unconnected to surrounding object representation. By changing voxel values one can manipulate a volumetric object, however in surface based graphics, a subset or the entire tessellated mesh would have to be recomputed. A rasterized representation is analogous to the way images are displayed to a user, continuous surface representations must first be rasterized before display. Thus volume approaches are well defined for intuitive high quality images, modelling arbitrary primitives and animation.

Winter and Chen [WC01] introduce a software based volumetric primitive API *vlib* for use in volume visualisation and volume graphics modelling. This API can be used to model constructive volume geometry, spatial transfer functions and other volume modelling techniques by treating volume objects as scalar fields. It provides an environment to explore volume datasets for volume visualisation and graphics and provides flexibility for volume modelling.

2.7.1 Distance Fields

Distance Fields or distance volumes are volumetric datasets that encode distances to structure within the objects domain. Distance fields represent a flexible tool in volume graphics to effectively add surface based representations, compute efficient volume metamorphosis and allow image-order rendering speed-ups. Distance fields can also be used to encode iso-surfaces from other voxelised data. There are other further uses for distance fields, such as examining differences between triangular meshes. Jones *et al.* [JBŠ06] provide a detailed review of distance field uses in computer graphics.

Distance fields are sets that encode the euclidean distance to a point on the objects surface within its domain. In some cases the gradient of the distance function can also be utilised to derive a direction toward the surface. When using signed distance fields, it is possible to ascertain if a point is outside the object, inside the object or on the surface of the object. The distance fields covered here are signed rasterized approximations of a continuous distance function. Equation 2.24 defines the distance field as a set.

$$D : \mathbb{R}^3 \rightarrow \mathbb{R} \quad (2.24)$$

$$\mathbb{D}(p) = \text{sgn}(p) \cdot \min \{|p - q| : q \in S\}$$

$$\text{sgn}(p) = \begin{cases} -1.0 & \text{if } p \text{ outside object} \\ 1.0 & \text{if } p \text{ inside object} \end{cases}$$

where $p \in \mathbb{R}^3$ is a point in the 3D grid throughout the objects domain, S is the set of points on the surface of the object and $|p - q|$ is the euclidean length of the vector between point p being sampled and q a point on the objects surface.

Rasterized distance fields can be computed with a variety of methods [SB02, GPRJ00, Jon96, SJ01]. The brute force algorithm for computing discrete distances is computationally very expensive, each point on the objects surface must be considered for each voxel location in the rasterized distance field. These approaches intelligently use neighbouring computed distances to build up the distance field and reduce complexity.

Voxelisation of surfaces and functions can be performed to yield a rasterized distance field volume, thus surface representations and functional representations can be rendered with DVR techniques. The complexity of highly complex surfaces and functions can be reduced by using the volume rendering approach.

In image-order algorithms, observing that normal rasterized volume densities can be empty and disregarded, distance information can be employed to space leap. If a distance value encountered at a sample location along the ray does not contribute to the surface, this distance can be skipped safely because it represents the distance to the closest point on the surface. [Šrá96, ŠK00]

Distance fields are also well defined for gradient normals. Conventional central difference (see section 2.4) can be used to generate gradient normals for iso-surface shading.

Distance fields will be explored later in this work for advanced texture synthesis and acceleration techniques.

2.7.2 Global Illumination

Global illumination is the computation of light transport through a scene to derive how a point in space is lit. Global illumination will usually include multiple light sources, reflection, refraction and shadowing effects. Yagel *et al.* [YKZ91, YCK92] introduce a raster ray-tracer (RRT) to ray trace volume densities. They build on standard ray-casting to include recursion analogous to surface based ray-tracing techniques. Rays are cast into the volume using image-order techniques and on discovery of a non transparent voxel, rays are spawned for reflection and towards each light source. Sobierajski and Kaufman [SK94] refine the original RRT algorithm to include other representations such as surfaces and add *radiosity* based rendering to the volume model.

Global illumination remains a complex problem to compute in volume representation due to the complex emission-absorption model employed to describe light transport though the volume dataset. Surfaces benefit from only considering light at a point on the surface and do not require modelling the light though the interior of the object. Ray-tracing algorithms yield the best possible images as they closely model real world physics.

2.7.3 Shadows

Shadows in volume graphics have been defined in work covered in section 2.7.2, however shadows have also been defined for locally illuminated DVR algorithms. The premise of adding shadows to a ray-casting algorithm is to fire a single ray at each sample point towards any light sources. If there are non transparent samples encountered, the sample is in shadow. The magnitude of the shadow is dependent on accumulated opacity between the light source and sample point.

Behrens and Ratering [BR98] add shadows to object-order texture based volume rendering on GPU's by reconstructing 2D planar volume slices. The reconstruction works separately to visualisation and produces a new volume dataset for rendering. Firstly a slice is taken, and the following slice is also taken with an offset defined for the lighting position. These two slices are then blended into the frame buffer. Each slice through the volume is processed until a shadow volume is produced. Recalculation of the volume is required when the light vector is changed. The rendering of the shadow volume is not described with lighting, additionally orthographic projections must be used and only one light source is considered. The reported speed is half that of normal rendering when calculating the shadow volume.

Nulkar and Mueller [NM01] also employ a 3D shadow volume representation however render with the original volume and blend with a lookup into the shadow volume during rendering.

Zhang *et al.* [ZC02] describe shadowing volumes using splatting. This approach is to keep a shadow buffer to describe accumulated opacity from the light source. Processing is carried out on a slice by slice basis in software. They compute contribution at a sampled point from the light sources with a shadow buffer and viewpoint with standard volume rendering. At each sampled point the shadow buffer is blended into the viewpoint slices final image. Both buffers are composited slice by slice as rendering continues. They describe multiple lights using multiple shadow buffers. The reported speed is under half the rendering of the volume for computing the shadows.

Kniss *et al.* [KPHE02, KKH02] also employ a 2D shadow buffer, however they change the orientation of each slice being rendered to halfway between the eye and light vectors. this allows simple compositing by observing that spatial coherence between buffers is maintained.

2.7.4 Constructive Volume Representation

Complex objects or scenes can be built up using a variety of simpler objects. Volume visualisation concentrates on rendering single volume datasets to produce a final image. In volume graphics it is desirable to combine objects to build up more complex objects or scenes to render. This ability is fundamental to any graphics representation that wishes to enable modelling and rendering of arbitrary shapes and collections of objects. These constructions can be modelled using trees where the root node represents the final scene description. These trees are often referred to as a *scene graph*. Leaf nodes of the tree encode definitions for simple objects and nodes define operations to apply to its branches.

A commonly used surface representation for construction is *constructive solid geometry* CSG, introduced by Requicha and Voelcker [RV77, Req80]. This representation defines boolean operations for solid construction of objects from surfaces, however the binary domain does not adequately allow construction of volume primitives since internal and external structure need to be included.

Wang and Kaufman [WK93] voxelise geometric primitives such as CSG objects in to a volume dataset. Objects are voxelised into one volume dataset for rendering. Anti-aliasing schemes for voxelised geometries are discussed to better approximate original geometry rather than blur jaggy edges for smoother display. This method can produce large sparse datasets in the event that objects are small and far apart.

Breen *et al.* [BMW98] discuss scan-converting CSG trees into distance fields for rendering. They employ distance fields for comparison of CSG representations and to use distance field modelling techniques such as morphing.

Fang *et al.* [FSV98] extend CSG operations to the volume domain and introduce *volumetric-CSG* (VCSG) by allowing operations on the real and integer domains instead of simply the boolean domain. Some boolean operations are redefined for real domain operation, however there can be different interpretations of their original meaning. They also suggest that constructing a single volume dataset to encode the VCSG tree is time consuming and does not allow interactive change of the tree. They solve this problem by subdividing the target dataset space and render sub-volumes for projection.

Constructive volume geometry (CVG) introduced by Chen and Tucker [CT00] provides an algebraic framework for describing volume construction using scalar fields. This representation provides a mechanism to apply operations to any field attribute in 3D environments. A small set of operations are defined for a volume datasets spatial, colour and opacity properties. These operations can be used to build up complex objects at the field level. These objects can then be positioned in a CVG tree. This modelling approach allows inclusion of *f-rep* or implicit surfaces and rasterized volume data such as distance fields or conventional scalar densities.

Chen [Che05] later introduced point clouds to the volume graphics pipeline using CVG. This method can include vertices from surface meshes to the volume rendering paradigm which gives surfaces a thickness and allows interaction with other volume primitives. Point clouds are regarded as scalar fields with attributes for spherical radius. Recently a large body of graphics research concentrates on the point as a display primitive. Levoy [LW85] originally highlighted the use of points as a fundamental display primitive.

2.7.5 Deformation and Animation

Volume rendering algorithms and volume modelling techniques can be used to render true 3D primitives, however their adoption as a general graphics primitive requires intuitive deformation and animation techniques. There are many surface methods that exist to deform polygonal meshes or parametric surfaces, these methods concentrate on moving vertices in the polygonal mesh before any rasterization. This can be counter intuitive since a collection

of points contained within one triangle can not be moved without changing the representation of the mesh. Since the volume is already rasterized, individual voxels in the volume can be changed without affecting the overall structure. This provides a mechanism for intuitive deformation such as sculpting and carving. Models can also be employed to allow deformation in a structured manner. Animation is closely linked to the ability to deform the structure of an object.

He *et al.* [HWK94] describe *volume morphing* using wavelets. Morphing involves calculating intermediate steps for transforming one start volume into an end volume smoothly. The use of wavelets allows smooth transition between models since high frequencies in the data can effectively be removed. Interpolation schemes are used to control the wavelet morphing function. Wavelets are used to attempt correspondence between the two volumes and no user interaction is required. This method does not allow control of the morphing for arbitrary properties such as colour and texture, control of features is also not well defined.

Lerios *et al.* [LGL95] describe *Feature-based volume metamorphosis*. They utilize warping with control points in the start and end volumes to allow features to morph into one another and blending is used to interpolate voxels between the target volumes for each frame. This method requires a good deal of user interaction. Control of the morph is possible in this manner and smooth results are presented.

Gibson [Gib97] describes a physically based strategy for deforming volumes based on 3D chain mail. A chain mail lattice is encoded through the volume by using a 6 connected voxel neighbourhood. Expansion, compression and relaxed states are modelled in each neighbourhood to allow fast deformation. Moving a chain in the lattice will deform the volume by changing the distance between voxels in a neighbourhood.

Gagvani and Silver [GS01] introduce skeleton based volume animation. The process of volume thinning [GS99] produces an unconnected set of voxels describing the skeleton of a binary segmented object. These voxels are then connected to form the skeleton. Distance field information is used from skeleton voxels to describe the original object. This skeleton is then deformed with existing animation techniques and packages. A volume is generated in respect of the deformed skeleton using distance information and rendered using standard volume rendering techniques.

Chen *et al.* [CSW⁺03] describe *Spatial transfer functions* as a framework for modelling deformation. This framework is capable of encompassing several deformation methods as described above. The application of the spatial transfer function is performed for a point before interpolation and classification. This allows arbitrary deformation and animations of volumes by enabling on the fly transformation operations or pre-computing the result of a deformation technique as a spatial transfer volume:

$$\Psi : \mathbb{E}^3 \rightarrow \mathbb{E}^3 \quad (2.25)$$

$$\mathbb{V}'(p) = \mathbb{V}(\Psi(p))$$

Where Ψ can be defined for sub functions as scalar fields $\Psi_x, \Psi_y, \Psi_z : \mathbb{E}^3 \rightarrow \mathbb{R}$ allowing individual field attributes to be represented separately:

$$p' = \Psi(p_x, p_y, p_z) = \begin{pmatrix} \Psi_x(p_x, p_y, p_z) \\ \Psi_y(p_x, p_y, p_z) \\ \Psi_z(p_x, p_y, p_z) \end{pmatrix}$$

Islam *et al.* [IDSC04] demonstrate splitting operations over scalar fields and spatial transfer functions. Implicit and explicit split operations are defined to enable splitting on attribute fields such as volume density and user defined explicit splitting with some user interaction. A splitting operation can result in multiple sub chunks of the original volume data and individual processing of their spatial transfer and volume rendering steps. They outline several operations for controlling the spatial location of split objects such as translation, rotation and scaling.

Walton and Jones [WJ06] introduce *volume wires* as a method of defining skeleton deformations using distance information to relocate samples. This method is computed on the fly during rendering and thus avoids a reconstruction of the volume for each deformation. The volume wire is defined using control points and encodes a skeleton of voxels. The wire is moved to perform deformation and new distance values are computed for rendering in respect of the original control wire. These distance values are used during rendering to relocate voxel positions and perform the deformation.

2.7.6 Procedural Texture

Procedural texture [EMP⁺03] can be evaluated over different domains and involves defining a function or collection of functions that operates in the desired domain. The discussion here is limited to a 3D domain which is also viewed as the solid texturing domain. Solid texturing describes the carving of a substrate material with an object's definition to produce the notion of a solid object. A Solid object is therefore effectively fashioned from a block of raw material. This material is generally defined procedurally using fractal contributions to model real world materials, although is not limited to procedural definition. Figure (see Figure 2.18) shows a procedurally generated marble substrate which is used for solid texturing a sphere's surface to represent a solid marble sphere. Procedural texture synthesis has also been a focus of providing *shaders* which allow definition of object surface properties and lighting conditions.

Blinn and Newell [BN76] discuss procedural techniques using Fourier synthesis. Simple wave and bump patterns are generated using Fourier transforms to apply image processing to a 2D texture image. These techniques produce computer generated images with natural appearances and represent early consideration of procedural techniques.

Fu and Lu [FL78] proposed a syntactic grammar for describing procedural texture. This allowed texture to be described with a texture language that includes arbitrary functions.

Fournier *et al.* [FFC82] utilize Brownian motion to describe stochastic variations of surface properties. The original Brownian motion computations are expensive and an approximation is presented to allow a small footprint function for evaluation.

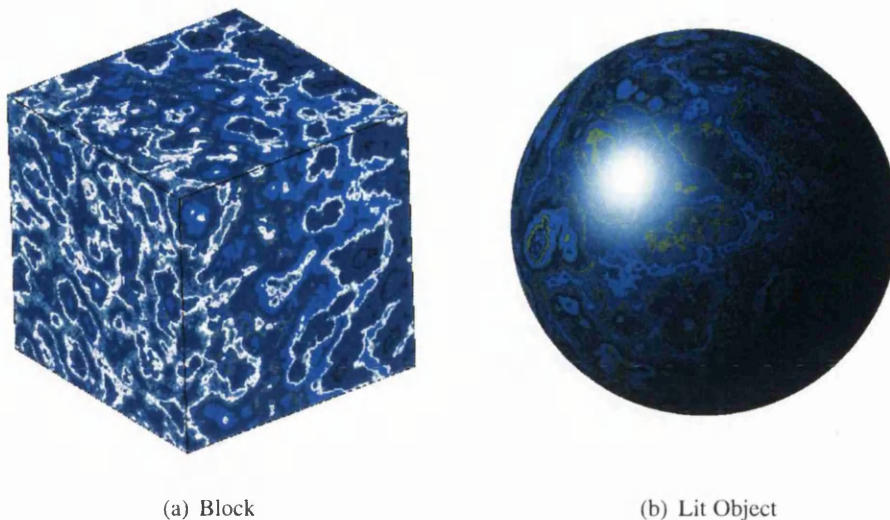


Figure 2.18: Solid texture block and solid textured object

Cook [Coo84] introduced *shade trees* as a method to describe texture, lighting, material and object properties as a specification of rendering parameters. This method of defining a tree structure to describe a complete set of surface properties allows simple inclusion of procedural techniques at any level and provides an extremely flexible modelling and rendering framework.

Peachey[Pea85] and Perlin[Per85] independently describe solid texturing which is an extension to basic 2D texturing which instead considers a 3D volume texture which represents a solid material or substrate. 2D texture synthesis over 3D primitives involves computing texture patches from a 3D position into a 2D texture lookup table or procedural function. There are a variety of sampling and filtering functions to compute this mapping, however most suffer from aliasing and are prone to introducing artifacts (Some 2D texturing methods are explored later in this thesis). This is due to the texture having to be wrapped or projected onto the object's surface. Both authors describe computing procedural synthesis for solid texturing.

Perlin [Per85] introduces *noise* (Perlin noise) along with a complete procedural texture generation language. Noise is a function that generates pseudo-random band limited white noise over a number of dimensions. The approach to providing a stochastic function is to encode a regular grid through the texture domain and provide pseudo-random gradients at each grid point. The scalar value at grid points is encoded as zero and internal grid point scalars are derived from interpolating along the random gradients at each grid corner. A cubic interpolation function is used to describe a complex waveform within the grid's cell. The corner contributions are then linearly interpolated for the final noise scalar. Perlin later improved on his noise implementation [Per02] to iron out artefacts introduced whilst cubic interpolation is performed for each pseudo-random gradient.

Lewis [Lew89] provides a comprehensive review of noise functions and suggests improvements and good qualities of several established noise algorithms. Additionally two algo-

rithms are presented that are more efficient and allow finer control than reviewed algorithms.

Hanrahan and Lawson [HL90] describe the RENDERMAN shading language, a descendant of the shade trees model used widely in software graphics pipelines for describing surface properties, texture and lighting parameters for rendering. These language definitions have formed the basis for many modern programmable shader languages.

Ward [War91] implements a lattice based noise implementation similar to Perlin noise, however replaces the pseudo-random element of the function to be a hash function. The computation is recursive and the performance of this method is improved, however the visual results of this noise function are not visually comparable to Perlin noise which is the target application.

Rhoades *et al.* [RTB⁺92] describe a real-time procedural texturing language and implementation for high-end graphics workstations. This implementation is an early precursor to more modern commodity hardware GPU implementations.

Westermann and Ertl [WE98a] use solid volumetric textures to texture polygonal meshes and additionally volume datasets are considered for future GPU implementations. GPU hardware was not capable of flexible vertex shading which required any display lists to be re-compiled before upload to the GPU memory to change texturing coordinates. The outlined method instead renders the vertex object space coordinates into the frame buffer as colour values with no lighting. The frame buffer is read back to main memory and passed into the GPU as a texture map. A second pass renders these colour values back into the frame buffer with pixel texturing enabled. The colour values are then used as texturing coordinates to perform a dependent texture fetch for each fragment. The solid texture space can then be manipulated with the colour matrix to perform transformations. These operations can now be performed on the fly with vertex and fragment shaders.

Hart *et al.* [HCK⁺99] define procedural solid texturing for GPU architectures. This model performs texture generation on the fly with pre-processing of texturing primitives. The object to be textured is then parameterised to map 3D object space coordinates of each vertex into a 2D texture map in hardware. Another pass performs fragment shading for the geometry and each fragment is textured with the parameterised texture map. Seams and aliasing can result from the 2D parameterization and filtering methods are provided to avoid discontinuities. The parameterization into a 2D texture map benefits from avoiding the host bus transfer speed whilst uploading large solid texture volumes. Initial results are based on most of the pipeline being performed in software due to limiting hardware. Carr and Hart [CH02] later performed these techniques in hardware with the addition of mip-map generation ability.

Hart [Har01] implemented Perlin noise on earlier GPU's before flexible fragment shaders became available. Multiple passes are used to implement a lattice based noise model by rendering solid texturing coordinates as colours and then adjusting the output colours in subsequent passes to compute the integer points, the linearly interpolated fractional parts and the lower front corner and upper back corner of the lattice. Applying a dependent texture into a random texture map and interpolating between lattice corners provides the stochastic noise effect. Whilst this method is transferable to more modern fragment shaders, the method is not very efficient due to the large multi-pass overhead. Additionally the output

noise is visually not as impressive as Perlin noise.

Satherley and Jones[SJ02] introduce solid texturing for volume objects in software. They perform binary segmentation on a volume object to obtain an iso-surface definition and colour the sample using procedural techniques on the fly. Lighting is then performed to produce a solid textured volume object. This software implementation performs the volume rendering pipeline in full without pre-computation but does not achieve real time results.

Mark *et al.* [MGAK03] introduce Cg as a high-level shading language to program GPU's. Cg allows real-time performance for shading languages and is capable of controlling most of the overall GPU pipeline. Procedural texturing techniques can be evaluated directly on graphics hardware using this approach.

Rushmeier *et al.* [JDR04] have demonstrated that it is possible to closely model naturally and man made materials by studying their construction. 2D cross sections of materials are examined in order to segment possible structures. These structures are then graded by shape, size and spatial relationship. A solid texture can then be derived using the spatial information with pseudo-random placement. This gives rise to being able to model many more materials such as concrete, stone and rock. Highly detailed natural looking images can be synthesised in this manner.

Green [Gre05] describes a fragment shader implementation of Perlin noise for GPU hardware. Texture maps are used to provide pre-computed lookup tables containing the pseudo-random function and the pseudo-random gradients. The function is computable in one pass and accelerations are introduced into the original algorithm to take advantage of single cycle vector arithmetic and additionally reduce texture lookups. This represents the best available noise function for single pass techniques on GPU hardware to date and provides a reference implementation of Perlin noise which generates visually superior results.

2.7.7 Hypertexture

Perlin and Hoffet [PH89] extended the solid texturing paradigm to produce hypertexture, a space filling texturing approach allowing modelling of natural phenomena such as fur, fire and smoke. Hypertexture is a method of manipulating surface densities whilst rendering, instead of evaluating colouring at the local surface. The actual surface definition is changed during rendering since a soft-region outside the original object surface is considered malleable and deformable with procedural synthesis. Because of this rendering outside the conventional surface definition, a ray marcher must be used. This does not suit the surface based graphics pipeline as a ray marcher must be implemented to provide hypertexture effects in complex scenes. An implicit surface is used to overcome the surface paradigms infinitely thin surface model.

Hypertexture is modelled with an object density function (see Eqn 2.26), giving rise to the notion of a soft object, or object which has a surface with depth associated. This gives three possible states to define an object:

- Inside - The point is inside object
- Outside - The point is outside the object and soft-region

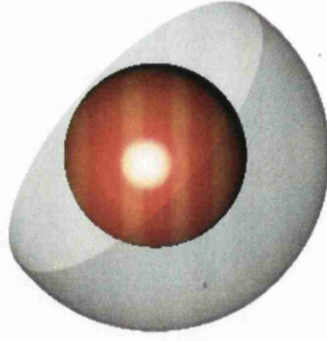


Figure 2.19: Sphere with object density function defining a soft-region. The soft region is clipped to show its relationship with the object

- Boundary - The point is in the soft-region or soft surface.

The following object density function models the implicit surface of a sphere with a soft-region. Figure 2.19 shows a graphical representation of a object density mapped sphere with a clipped soft-region.

$$D(p) = \begin{cases} 1 & \text{if } f(p)^2 \leq r_i^2, \\ 0 & \text{if } f(p)^2 \geq r_o^2, \\ \frac{r_o^2 - f(p)^2}{r_o^2 - r_i^2} & \text{otherwise.} \end{cases} \quad (2.26)$$

where r_i = inner radius, r_o = outer radius and $f(x)$ is the sphere radius function .

Hypertexture effects can now be achieved by the repeated application of density modulation functions (DMF functions) to the soft-region of $D(p)$, as shown in Eq. 2.27.

$$H(D(p), p) = DMF_n \left(\dots \left(DMF_0(D(p)) \right) \right) \quad (2.27)$$

The base DMF functions are the functions *bias* and *gain* which are used as control curves, *noise* and *turbulence* to create pseudo-random patterns and general mathematical functions including as periodic functions such as *sin* and *cos*. Higher order DMF functions are defined with these base primitives for manipulating the soft-region of a hypertextured object. DMF functions can be characterised into the following groups.

- Position dependent - functions which depend on p
- Position independent - functions which exhibit scalar arguments
- Geometry dependent - functions which require local surface geometry (e.g. gradient normals)

Bias is introduced to hypertexture as base a function for DMF functions (see Eqn 2.28). Figure 2.7.7 depicts differing values for the bias function and the resulting curves. DMF

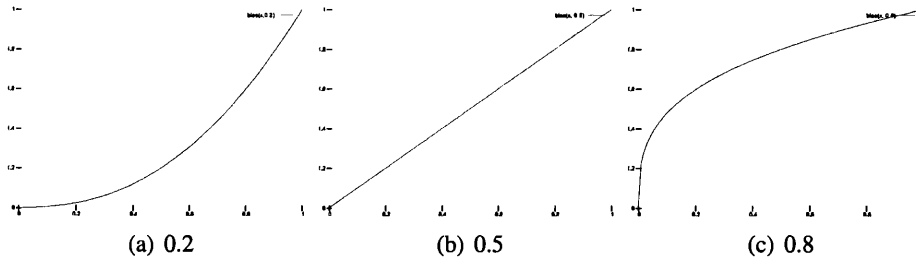


Figure 2.20: Bias curve functions with differing b values

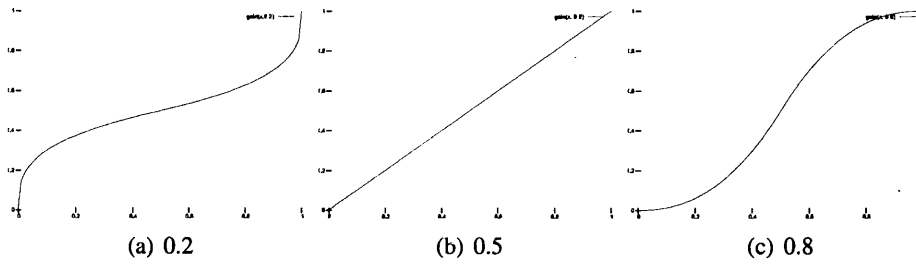


Figure 2.21: Gain curve functions with differing g values

functions are linear across the width of a soft-region. Bias can be used to alter the linear nature across a soft-region, power curves are used to focus attention to a particular area within a soft-region.

$$\text{bias}_b(D(p)) = D(p)^{\frac{\ln(b)}{\ln \frac{1}{2}}} \quad (2.28)$$

Gain is introduced to hypertexture as base a function for DMF functions (see Eqn 2.29). Gain is built up from two bias functions to control the midrange rate of change in the soft-region (see Figure 2.7.7).

$$\text{gain}_g(D(p)) = \begin{cases} \frac{\text{bias}_{1-g}(2D(p))}{2} & \text{if } D(p) < \frac{1}{2}, \\ 1 - \frac{\text{bias}_{1-g}(2-2D(p))}{2} & \text{otherwise.} \end{cases} \quad (2.29)$$

Both the bias and gain functions are also useful for altering the output of *noise* functions and control their statistical and spectrum compositions.

Satherley and Jones [SJ02] introduce hypertexture for distance fields which removes the restriction of simple implicit surfaces. They procedurally generate the soft-region with noise primitives and built up DMF combinations in software. The object density function is replaced to define a soft-region from a distance field volume, the inner radius representing the chosen iso-surface and the outer radius being the chosen soft-region boundary. In this manner complex volumetric objects can be hypertextured. The object density function replaces the standard binary segmentation performed on volume datasets for iso-surfacing. These techniques however are not achieved in real-time.

2.7.8 Texture Mapping

There are many well known 2D texture mapping techniques that improve the visual aesthetics and complexity of a surface definition for surface graphics. These methods are utilised to allow more complex looking surfaces without impacting rendering performance by re-forming, refining or re-modelling the entire polygon mesh. A uv parametrization of the surface is required to address the texture space, which is generally the unit square.

Catmull [Cat74, Cat75] and Blinn [Bli78a] introduce texture mapping as a technique for adding colour information to an object without increasing the object's geometric complexity. Colours are contained in an image map from any source capable of defining a 2D domain. This allows digitized photographs or output from 2D image modelling and drawing packages to be applied to an object's surface. Images obtained from this technique appear flat as this method does not include detailed physical properties from the underlying colours such as reflectance and gradient.

Textured objects are obtained by computing parametric patches in texture space by assigning texture space co-ordinates to each object vertex. These texture space co-ordinates are interpolated over an object primitive such as a triangle or polygon to obtain texture space co-ordinates for each point on the surface defined in screen or pixel space. Aliasing can be introduced since the texture map's resolution does not accurately match that of the screen space. Methods such as weighted contributions are discussed to avoid these issues. Using this technique requires objects that can be assigned a uv parametrization and fall short of defining texture on all objects.

Williams [Wil83] introduced mip-mapping (see Figure 3.5), a technique to describe level of detail texturing. Objects in rendered scenes can be subject to differing magnifications depending on a scene's construction. Artifacts and aliasing can be introduced under magnification and minification since the texture space can be addressed at differing frequencies. Mip-mapping avoids artifacts, aliasing and additionally performance degradation by encoding the original texture resolution at the root level with progressively half sized textures defined at subsequent levels. The images in these levels are generally processed in some manner to avoid aliasing and artifacts. The original algorithm describes 2D texture space, however any dimensional space may be used.

Bier and Sloan [BS86] introduce projected texture mapping, a proxy geometry texture mapping algorithm that projects in a direction away from the objects surface to an intermediate simple geometry with known parametric properties. Boxs, spheres, cylinders and planes are described as proxy geometries and a mapping is formed between a 2D texture map and 3D object by projecting onto the intermediate surface which contains a uv parameterization. This effectively shrink wraps a texture map around an object. A reverse of this mapping process can also be carried out in order to make texture templates to allow texture artists a 2D version of the object to paint on. This method solves the previous problem of applying a uv parametrization to objects without well defined surfaces by choosing a known intermediate representation.

Winter [Win02] noted the importance of including surface graphics texturing techniques for volume datasets and covered projective texture mapping. Winter and Chen [WC01] describe

vlib where 2D and 3D texture mapping are features of the API in software. The projected texture mapping algorithm is used for 2D texturing operations since volume datasets cannot be parametrised for a 2D texture space.

Shen and Willis [SW05] describe the use of 2D projective texture mapping algorithm for volume datasets. They additionally demonstrate level of detail, tiling and arbitrary positioning the texture over a volume objects surface in software.

2.7.9 Bump Mapping and Displacement Mapping

Blinn [Bli78b] introduces *Bump mapping* which allows a smooth uniformly defined surface to include bump information without increasing the complexity of the underlying object. The algorithm displaces surface normals with a bump function or height field which describes a height of pseudo displacement for a point on the surface along the surface normal. New normals for this proxy surface are then obtained and used in the lighting computation. This method produces a bumpy object surface however the perturbed effect is not defined at the silhouette of an object since no geometry is altered.

Peercy *et al.* [PAC97] used a pre-computed normal map to speed up rendering in hardware and avoid the expensive normal computations involved per fragment. Two techniques are presented that pre-compute displaced normal vectors from a monochrome height field. The scalar differences are used to pre-compute normals in texture or tangent space. Graphics hardware is used to map each surface normal from object space into tangent space, allowing fast replacement of tangent space pre-computed bump normals. Additionally object space bump maps are computed such that the surface parametrization is taken into account during pre-processing. This allows direct replacement of object space normal vectors. The lighting is described in object or tangent space to avoid mapping each displaced vector into eye space for lighting.

Kilgard [Kil00] discusses hardware implementation of several bump mapping algorithms. Tangent space and object space are discussed for efficient lighting of surface mesh bump map techniques. Pre-computed normal maps are used to accelerate rendering on GPU hardware. In addition implementation specific detail is presented to allow the efficient tangent space and object space normal mapping techniques to be realised on older GPU architectures.

Cook [Coo84] introduces *Displacement mapping* which solves the problem of rendering a bumpy or wrinkled surface at the silhouette of an object, however this algorithm is significantly more computationally expensive. Generally a surface based wire frame mesh is displaced with a height field which is analogous to bump mapping. Since displacement mapping is performed before rasterization the complexity of the algorithm is described by the underlying object representation.

Hirche *et al.* [HEGD04] introduce a post-rasterization displacement mapping technique. Extra geometry is rendered around the object to displacement map and each pixel from the resulting geometry is used to ray-cast into this proxy displacement geometry in the fragment shader. Complex intersection equations evaluate if the fragment is part of the object's displacement and coloured accordingly. This method benefits from not having to

specify level of detail since fragments considered perform this step and additionally no object geometry is dynamically changed during rendering.

Wang *et al.* [WWT⁺03] use a height field to generate a view dependent displacement function. This displacement function records the distance to the reference surface along the viewing direction with an additional surface curvature term. This 5D lookup table or function is computed by firstly rendering a highly detailed surface mesh representing the displacement. This object is then subject to ray-casting with different viewing parameters to define the view dependent function. The simplified surface is then subjected to fragment shading and inward displacements are calculated from a uv parameterization of the surface, viewing direction and curvature term. In this manner no geometry is changed or refined during rendering.

Wang *et al.* [WTL⁺04] expand their previous work [WWT⁺03] to provide a generalised displacement map that is capable of modelling structure that is not connected to the original surface. Previously displacements were restricted to emanating from the original surface. This approach allows arbitrary mesostructure to be defined using a volume displacement map. This method is still described as view dependent and a 5D function is required to sample the displacement map where viewing parameters are also required. The surface curvature is not modelled directly in this algorithm and is replaced with an additional texture space coordinate dimension describing height. The volume representing mesostructure is subject to ray-casting from differing viewing positions to determine a distance to mesostructure surface.

Porumbescu *et al.* [PBFJ05] define a displacement mapping mechanism that introduces further geometry into the rendering pipeline to define a displacement region and use geometric or volume procedural textures to fill this space. A tetrahedral mesh is utilised by directly encoding the outermost surface with the same construction. The space between these two identically constructed surfaces is then subject to tetrahedral division which guarantees a mapping from geometry to texture space where each tetrahedra has an individual disjoint region. This method therefore depends on the detail represented in the displacement region as more tetrahedral primitives are required for refined detail.

Max and Becker [BM93, MB94] later adjusted the original bump mapping algorithm to suit displacement mapping. This facilitated easy change between the two techniques for level of detail approaches, however this adjustment involved new terms which are more expensive to compute and adds further complexity into both bump mapping and displacement mapping.

2.8 Summary

In this chapter a thorough review of volume rendering techniques has been explored from their inception through important developments to current research. The whole volume rendering pipeline is defined with attention to surface reconstruction techniques and direct volume rendering techniques. Surface reconstruction is shown to provide binary segmentation at best and not include internal and external object detail. In contrast direct volume rendering disciplines are shown to exhibit fuzzy and binary segmentation and include internal and

external object detail, as well as amorphous and semi-transparent properties. Direct volume rendering has also been shown as a more intuitive approach to volume representations.

It is noted that volume visualisation techniques primarily used to visualise medical data have spawned an important sub field in computer graphics, namely volume graphics. Volume graphics is explored as a general graphics primitive with several important characteristics defined. Techniques to create, combine, deform, texture and animate volumes are explored as an analogous to important surface techniques, which highlights the importance of this representation as an intuitive and rich domain for general graphics computation.

A further observation has been noted concerning recent research achieving real-time display of volume data through the use of high-end workstation hardware and consumer level hardware. Hardware acceleration and direct volume rendering methods will be utilised for this work in later chapters.

Chapter 3

Direct Volume Rendering

Contents

3.1	Hardware Pipeline	50
3.2	GPU Volume Rendering Algorithms	60
3.3	Improvements	73
3.4	Comparison	80
3.5	Summary	109

Recent research in volume visualisation has involved the use of high-end graphics hardware and consumer level graphics hardware (GPU's) for rendering volume datasets in real-time. These graphics processors typically encompass multiple pipelines to compute graphics algorithms with surface base primitives in parallel. Use of GPU's can drastically reduce the runtime of rendering algorithms by exploiting this parallelism on a single workstation. GPU's also offer vastly increased memory bandwidth and specialized vector processing units and functions in hardware.

There are schemes for parallel rendering that utilize multiple processors, but are fixed to single processors on a single machines over a network. Parallel rendering in this manner can be fast, however the overall cost of such a system is vast in comparison to a specialised parallel GPU on one machine. If a volume dataset fits into GPU memory it is often faster to take advantage of the parallelism on a single platform due to network overheads and the expense of multiple machines. Specialist volume rendering hardware has also been developed [PHK⁺99], although this is restricted to high-end users due to cost and availability. Additionally no mixture with surface primitives is described which makes it inflexible for general graphics techniques with volume graphics techniques.

High-end graphics workstations with specialised graphics hardware [Ake93] and GPU hardware is geared towards surface based graphics and processes over primitives used in this approach. These hardware graphics pipelines share similar architectures but are available on differing machine architectures at drastically different costs. The high-end graphics workstation machines are considerably more expensive than consumer level GPU hardware and additionally represented the earliest means of encompassing hardware pipeline techniques.

These pipelines later became available on modern GPU hardware and the techniques detailed are transferable between platforms. Both architectures now exhibit extremely similar programmable pipelines although differences are evident which makes fully cross platform solutions infeasible.

The ability to mix surfaces and volumes in the same rendering environment is an advantage to using high-end graphics workstations with specialised graphics hardware and GPU's over more expensive specialist volume rendering hardware, additionally new features are added at a vast rate to GPU's which allow possible extensions of volume rendering techniques. GPU's are available on most modern machines as standard giving rise to techniques using this cheap and widely available hardware becoming predominant for development. The base primitives that a GPU processes are vertices's from a polygonal mesh which are rasterized, and the *fragments* resulting from rasterization. An overview of the pipeline between CPU and GPU is given in Figure 3.1 where vertex and fragments can be seen as the GPU's base primitives for computation.

The evolution of GPU hardware is rapid and new features are introduced with each generation. Some algorithms are not possible to implement on older generations of the hardware so an overview of the architectures and generation changes are detailed in section 3.1. The possibilities to define both image-order and object-order volume rendering algorithms on GPU hardware is explored in section 3.2. Improvements to the hardware based volume rendering techniques are detailed in section 3.3. A comparison of these approaches is detailed in section 3.4 with details of implementation. Finally a summary is given in section 3.5.

3.1 Hardware Pipeline

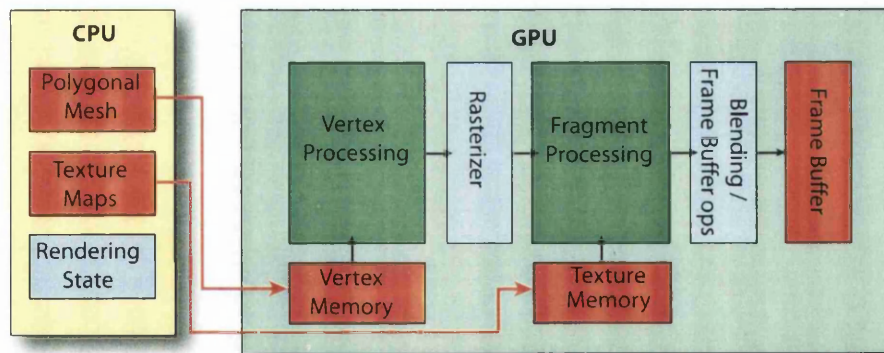


Figure 3.1: GPU pipeline overview - Red elements represent memory, green elements represent programmable units on some generations and blue represents fixed function configurable elements

GPU's were originally introduced into the graphics pipeline to offload and accelerate parts of the complete surface rendering pipeline. They have become more complex over time, allowing a rich set of new techniques to be included in real-time rendering. Real-time software

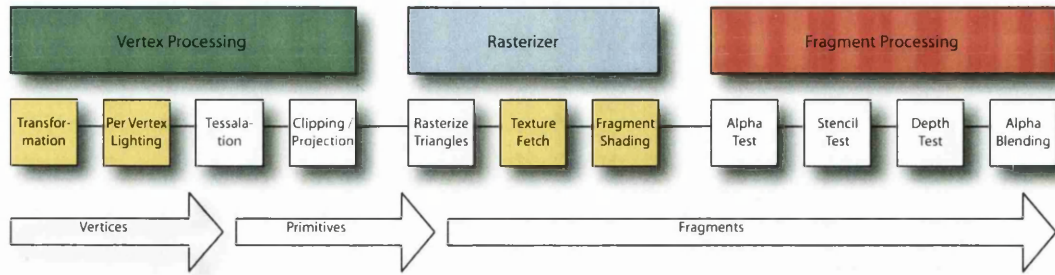


Figure 3.2: Generic hardware pipeline - Yellow elements represent programmable elements and white elements represent fixed function configurable elements. Arrows depict the primitives or representation at each point in the pipeline.

rendering is limited due to heavy processing constraints on vector types with architectures not suited to high throughput. Software techniques allow arbitrary techniques to be defined because of a more general architecture. However the emergence of programmable commodity graphics hardware allows adoption of some software rendering techniques to a hardware implementation allowing an increase in rendering performance. More algorithms available in software can be adapted to GPU's as feature sets grow allowing real-time rendering of more complicated scenes and more photo-realistic images.

Complex surface based techniques such as ray-tracing [Whi80] and radiosity [GTGB84] for large and complex scenes are still considered a software problem, although recent efforts have been made to allow GPU ray-tracing of small scenes with low complexity surfaces [PBMH02].

GPU's are stream processors that apply functions to their input in a continuous manner. The fundamental surface graphics pipeline is depicted in Figure 3.1. It demonstrates the boundary between CPU and GPU with reference to vertex lists and texture maps. The CPU sets state options to configure hardware resident rendering algorithms, builds a vertex array in graphics memory and uploads texture maps to graphics memory. The vertex array and texture maps are passed over an appropriate bus (usually accelerated graphics port (AGP) or Peripheral Component Interconnect Express (PCI-X)) to the graphics hardware for use during the rendering pipeline. Graphics memory is used to differentiate RAM which is resident on the main system and available to the main CPU and additional RAM available on the graphics hardware. The graphics memory in the graphics hardware must be either loaded or read after processing of the entire pipeline. Once loaded into the GPU vertex arrays are accessible from the vertex processing unit, whilst texture maps are accessible from the fragment shading unit. Recent hardware (5th Generation, NVIDIA 6800) includes the ability to allow texture map lookups during vertex processing.

For each frame, the vertex array is traversed per vertex with tessellation being derived from the configured rendering state. Each vertex is passed through the vertex processor and passed on to a rasterizer unit. At this point the tessellation of a collection of vertices from the vertex processor is used to rasterize the faces of the encoded geometry into fragments by scan converting from the image plane. These fragments are passed to the fragment processing stage where the result is eventually written into the frame buffer for display. The display device is directly connected to the graphics hardware and receives a frame buffer

for output.

3.1.1 GPU Generations

GPU's began as fixed function processing units in the graphics pipeline with the ability to transform and light vertices, rasterize polygonal meshes defined by vertices and texture these primitives by changing individual fragments according to a texture map. Functions have increased since the GPU's introduction, a table outlining generations and capability is given in Table 3.1. Differences can be better visualised by referencing Table 3.1 with Figure 3.1 which provides an overview of the architecture and Figure 3.2 which is more detailed. The overview in Figure 3.1 shows that vertex processors and fragment processors are programmable in some generations. Figure 3.2 expands this pipeline into individual elements and depicts which elements are programmable. Programmable fragment processing is seen as part of rasterization hardware but is referred to as fragment processing because of the primitives involved.

GPU generation	Capability
software	No hardware capability
1 st Generation	Rasterize geometry and texture
2 nd Generation	Transform and light vertices
3 rd Generation	Basic vertex programs and configurable rasterization
4 th Generation	Additional vertex instructions and fragment programming
5 th Generation	Vertex texturing and loops for fragment programs

Table 3.1: GPU generations and additional hardware capabilities

Each generation provides increased clock speed, richer instruction sets, additional parallel pipelines, increased on board memory and other features such as additional texture units, additional memory and additional memory bandwidth. There are two main API's for programming the complete graphics pipeline and utilising GPU's, DirectX [Mic06] and OpenGL [SA]. OpenGL is considered the best platform to develop GPU rendering algorithms as features are added when they become available and the implementation is cross-platform. Example GPU cards are presented in Table 3.2.

Generally vertex and fragment programs (*shaders*) are written in assembly language that is transformed into machine code for specific GPU's via a graphics hardware device driver. There are different sets of assembly language available and in OpenGL are defined by Architecture Review Board (ARB) [Ope] extensions to the standard API. Vendors of GPU's generally use their own assembly languages which are provided to the API via extensions. There are generalised assembly languages available via extensions that a vendors driver implementation can translate into its own native assembly language to provide cross-vendor shaders. New features are generally added in the native assembly language and filter down to generic assembly languages at a slower rate.

There are numerous higher level languages that can be compiled to different code bases for vendor specific and generic assembly languages. Mark *et al.* [MGAK03] describe C for

Generation	NVIDIA	ATI
1 st	GeForceTNT	Rage
2 nd	GeForce 2	Radeon 7
3 rd	GeForce 3 & 4	Radeon 8
4 th	GeForce 5	Radeon 9
5 th	GeForce 6 & 7	Radeon X

Table 3.2: Example GPU series

graphics (Cg) as a high-level language with C like syntax for compilation to GPU assembly languages. Cg is capable of compiling DirectX and OpenGL style shaders for a variety of vendors instruction sets. Kessenich *et al.* [KBR] introduce OpenGL shading language (GLSL) which is only available to OpenGL but compiles assembly code for a variety of vendors instruction sets. GLSL contains more functions than Cg and exhibits a tighter syntactic definition.

Table 3.1 shows that the 1st generation of hardware was only capable of rasterizing and texturing, the vertex processor would be omitted from Figure 3.1 for such capability. From the 2nd generation upward the whole pipeline in Figure 3.1 can be utilised to some degree by configuring the hardware fixed function algorithms state, however later generations exhibit more functions and programmable vertex and fragment processors using shaders. 3rd generation vertex processors are programmable with vertex shaders and 4th generation architectures upward allow programming of vertex and fragment processors with shaders.

It is possible to use GPU's for volume rendering from the 3rd generation up, and although later generations allow more complex algorithms to be defined in hardware. Although it is possible to define rendering algorithms on 3rd generation architectures, shaders provide a superset of configurations to be realized. Thus shaders are described in the remainder of this thesis as they are available on hardware used.

3.1.2 Data Types

Vertex and fragment processors operate on $1, \dots, 4$ component vectors and matrices ($1, \dots, 4$ element arrays of vectors):

$$\mathbb{F}_n : < f_1, \dots, f_n >$$

Where each $f_i \in \mathbb{R}$ builds up a n -tuple vector \mathbb{F} .

Vector components are arbitrarily selectable using a vector *swizzle*. An n component vector \mathbb{F}_n can be swizzled using:

- Identity: $< x, y, z, w > .xyzw \rightarrow < x, y, z, w >$
- Explicit ordering: $< x, y, z, w > .wzyx \rightarrow < w, z, y, x >$
- Component selection: $< x, y, z, w > .xxyy \rightarrow < x, x, y, y >$

- Smaller n -tuple selection: $\langle x, y, z, w \rangle .xz \rightarrow \langle x, z \rangle$
- Larger n -tuple selection: $\langle x, y \rangle .xyy \rightarrow \langle x, x, y, y \rangle$

GPU's can compute a 4-component vector operation in one instruction allowing a four fold increase on vector arithmetic. The stream based execution model also allows parallel processing of primitives since there is no dependency on one primitive to another. Additionally some vertex processors and all fragment processors can utilize texture maps from texture variables passed as a global. The output of the fragment processor will go through the fixed function fragment processing part of the pipeline and be output to the frame buffer for display.

3.1.3 Memory and Registers

Instructions for processing are downloaded to the graphics memory and fetched from processing stages into fixed instruction registers. These registers are not subject to any manipulation.

Vertex arrays are downloaded to graphics memory for use with fixed function hardware which delivers a single vertex description to the vertex processor via specialised input registers. A single vertex is fetched from graphics memory and loaded into these registers. This fixed function hardware delivers vertex descriptions in parallel to the available parallel pipelines defined for the vertex processor by multiplexing. After a vertex has been processed, the results are written into specialist output registers for presentation to the fixed function rasterizing hardware. Another multiplexer stage is used at this point since there can be differing numbers of vertex and fragment pipelines and tessellation of vertices must be considered. There are generally more fragment pipelines than vertex pipelines since a simple triangle will usually yield many more fragments than vertices. The fixed function rasterizer presents fragments to the fragment processor with specialist input registers analogous to vertex processing.

Each hardware implementation will have a fixed number of specialist registers that are available to define vertices in a vertex array and additionally a fixed number of specialist registers throughout the remainder of the pipeline. Additional general input registers are also available to allow global variables to be included. These registers are not writable within processing and are not available per primitive. There are also a number of temporary registers present in both processing units used for intermediate computations. The number of these additional input registers and temporary registers are also fixed for a specific hardware implementation.

Texture maps are downloaded to graphics memory for use in computations, these portions of memory are addressed directly when texture instructions are encountered and results are loaded into a temporary register.

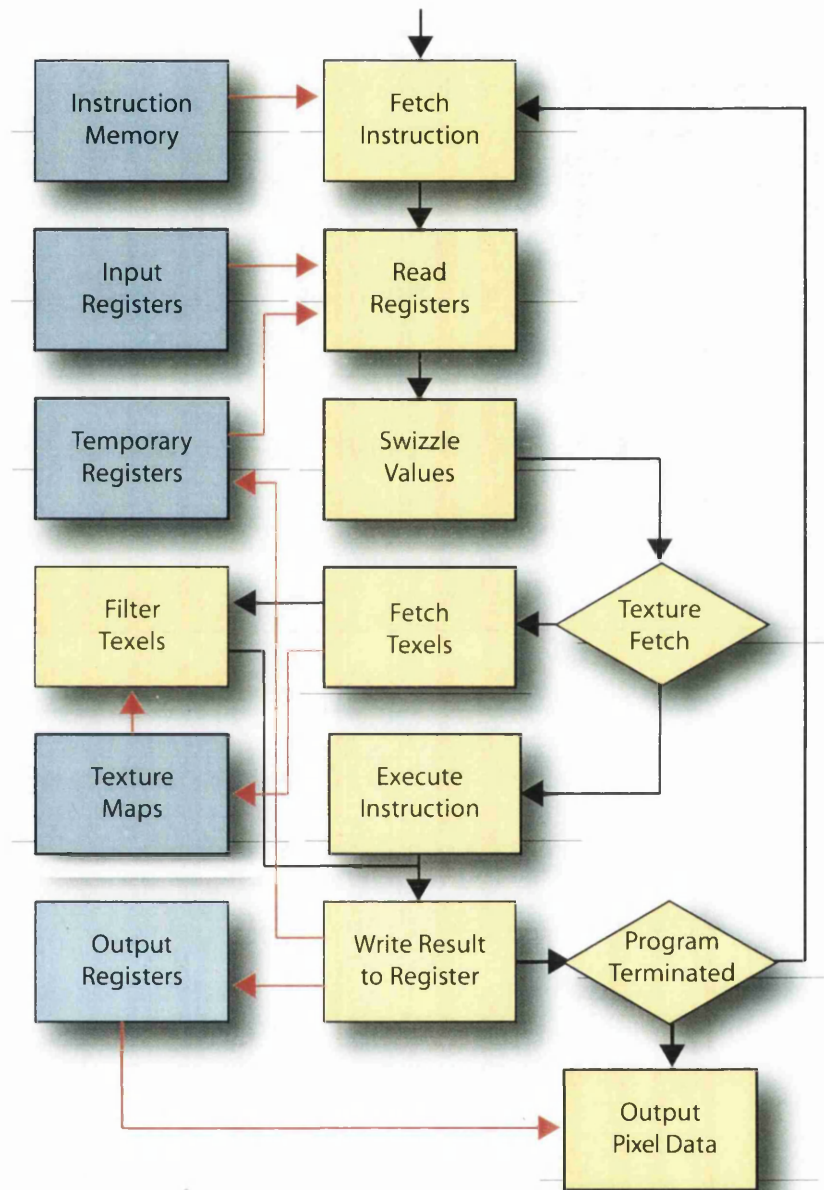


Figure 3.3: Vertex shader execution algorithm

3.1.4 Vertex Processing

The vertex processing unit is used to process each vertex individually using vertex descriptions resident in specialist registers. Additionally other registers can be used during computation.

A vertex description (resident in special registers) can contain:

- Position
- Colour
- Normal
- Texture Co-ordinates ($1 \dots n$)

Where $1 \dots n$ denotes the number of texture coordinates definable for each vertex on a specific hardware platform.

The vertex processor outputs a set of vectors into special output registers. The output of the vertex processor will be rasterized once vertex tessellation is computed and each special register will be interpolated across a tessellated triangles face for each fragment. These interpolated values are then presented to the fragment processor as special input registers per fragment.

- Position
- Colour
- Texture Co-ordinates ($1 \dots n$)

3.1.5 Fragment Processing

The fragment processing unit is used to process each rasterized triangle primitive that comes from the fixed function rasterizer. A fragment description is held in special input registers to the fragment processor.

A fragment description can contain, analogous to the vertex processor output:

- Depth
- Colour
- Texture Co-ordinates ($1 \dots n$)

A Fragment processor can output to special output registers:

- Colour
- Depth

The depth output register is used to update the optional depth buffer whilst the colour output register is used to update the frame buffer for display.

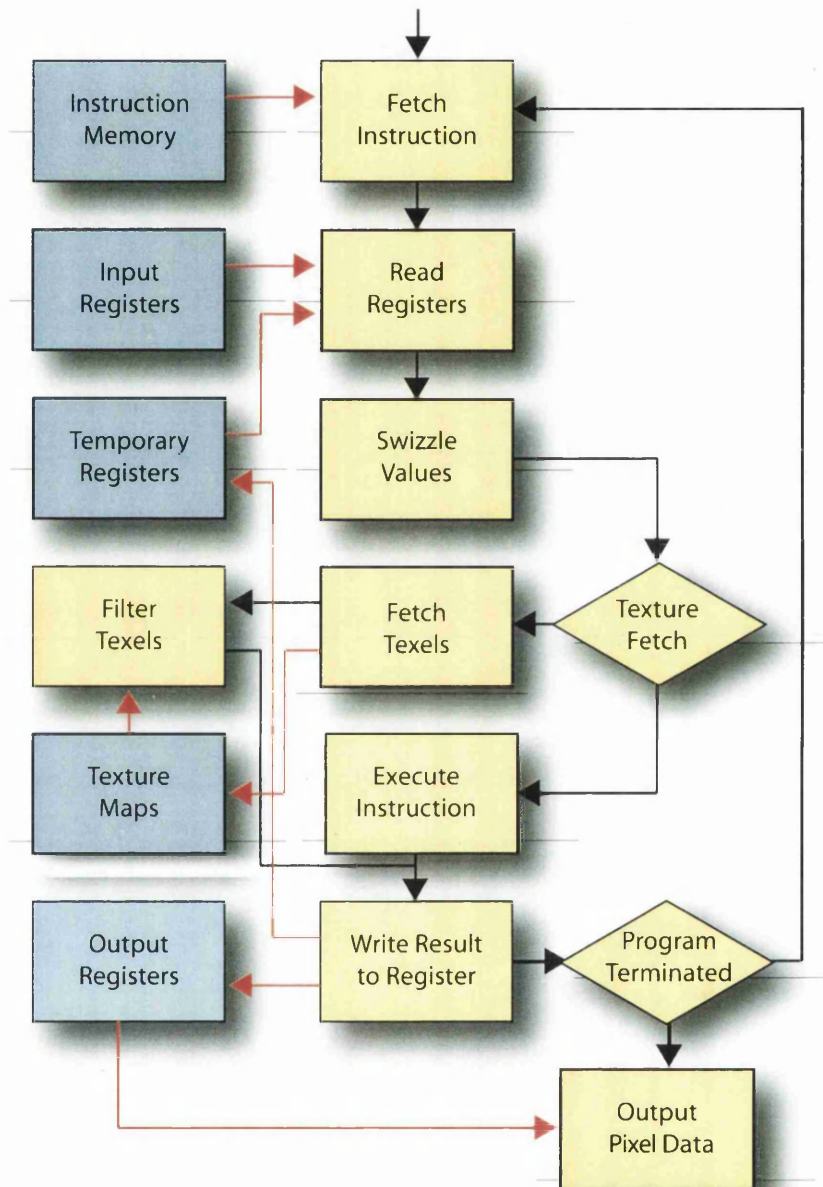


Figure 3.4: Fragment shader execution algorithm

3.1.6 Texturing and Buffers

Early GPU texturing units only address texture dimensions that are $\langle ds, dt, dr \rangle \in 2^n$ where $n \in \mathbb{N}$ for a 3D texture. These strict dimensions are defined for easy MIP-MAP generation and addressing. A mip-map is a recursive texture definition that at the top level encompasses the complete texture. Subsequent levels are half the size of the previous level to a defined smallest level. The mip-map texture is addressed with an additional argument describing the level of detail to address. Various operators can be applied to the texture map to compute each sub-level, the average, maximum and minimum values are common. Figure 3.5 depicts various mip-map levels. Additionally texturing units could only address 1D or 2D texture maps. Strict non power of two texture dimensions are not required on newer generations of hardware and 3D texturing is implemented.

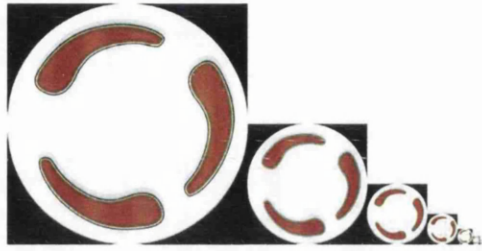


Figure 3.5: 2D Mip-map texture

Texture co-ordinates are defined as $\langle s, t, r \rangle \in [0, 1]$ or normalized co-ordinates. This allows differing texture modes to allow repeating textures, mirrored textures and clamped textures when the texture co-ordinates fall outside the $[0, 1]$ range. Later generations are capable of addressing the physical dimensions of a texture map directly.

A texture map is defined as a 1D, 2D or 3D regular grid. Each individual element is a *texel* (texture element) which is analogous of pixel and voxel for pixel elements and volume elements respectively. When a location in texture space is encountered that does not directly address the regular grid, two signal reconstruction filters are available in hardware. These are nearest neighbour and linear interpolation (see section 2.2).

Texel formats were originally clamped to the $[0, 1]$ range in hardware with fixed point scalars. A texture map with ranges beyond $[0, 1]$ has to be quantised. Later generations of GPU's allow arbitrary floating point scalars to be defined for each texel. A texel is limited to be a 1, 2, 3 or 4 component vector type describing a single scalar, an $\langle r, g, b \rangle$ colour or an $\langle r, g, b, \alpha \rangle$ colour. Remapping functions can be used to firstly encode a texture in the $[0, 1]$ range, denoted forward remapping (see Eqn 3.1) and reverse this mapping to obtain the original values (see Eqn 3.2).

$$t' = \frac{t - t_{min}}{t_{max} - t_{min}} \quad (3.1)$$

where $t' \in [0, 1]$, t is the original non normalised texel, t_{min} and t_{max} are the minimum and maximum texel values in the texture map.

$$t = t' (t_{max} - t_{min}) + t_{min} \quad (3.2)$$

A detailed table of texture unit capability is given in Table 3.3.

GPU generation	Non 2^n dimensions	3D textures addressing	outside $[0, 1]$ range	outside $[0, 1]$ texels
1 st	no	no	no	no
2 nd	no	no	no	no
3 rd	no	yes	no	no
4 th	no	yes	yes	no
5 th	yes	yes	yes	yes

Table 3.3: GPU generation texture unit capabilities

There are three special buffers defined in the regular pipeline. These are the frame buffer, depth buffer and stencil buffer. The frame buffer is ultimately used to capture an image and acts as input to a physical display device. The depth buffer is used to optionally reject fragments based on a boolean decision between the current fragments depth value (in image space) and the associated entry in the depth buffer (in image space). The depth buffer in surface graphics can be used for hidden surface removal [FvDFH96]. The stencil buffer is used to optionally reject fragments based on binary values in the stencil buffer. Usually these special buffers are the same resolution and are defined for a particular image space viewport. These special buffers can additionally be double buffered to allow asynchronous reading of one buffer and writing of the other. This allows a speed-up in rendering. The special buffers are not readable and writable in the same pass though the pipeline.

Auxiliary buffers in the graphics memory can also be created to divert fragment output into memory and not into special buffers. Depth and stencil buffers can also be created in addition to new general buffers. Auxiliary buffers are not readable and writable in the same pass through the pipeline. An algorithm that can compute an output image or frame in one iteration of the pipeline is labeled as a single pass algorithm. It is possible to encode multiple passes through the graphics pipeline without auxiliary buffers by leaving the frame buffer intact between passes. Auxiliary buffers are also used to collate intermediate information that is only possible to compute in one pass, where more processing is required to compute the remainder of a particular algorithm. Since the buffers are not readable and writable in the same pass, the buffer is uploaded to the machines main memory, constructed into a texture map and downloaded back into graphics memory for reading during a pass through the pipeline.

Recent GPU's (5th generation, NVIDIA 6800) have enabled reading and writing to a buffer without the need to upload the buffer contents to main memory and download this back to graphics memory. These buffers are still not readable and writeable in a single pass, however the buffer can remain in graphics memory and reading or writing the buffer is switched before each pass. This can accelerate multiple pass algorithms considerably since a pipeline stall is required to both upload and download from GPU memory. The host computer's bus speed defines the maximum speed at which reads and writes from the GPU are possible. If

a 0.5 GB per second download speed and a 2GB per second upload speed is assumed (AGP 8x bus) then an application requiring a read of a texture map in GPU memory (0.5GB in size) will save a second on every frame for each read since downloading the texture map is not necessary. Further savings are also possible since the upload is also no longer required.

3.1.7 Branching

Looping and conditional expressions pose a problem on GPU hardware as they involve branch instructions. The original GPU hardware is designed as a stream processor that does not suit out of order execution or branches in the source code. Newer hardware does include the ability to perform branches (see Table 3.1) in the fragment shader, however it is an expensive operation to perform.

Older hardware deals with looping by unrolling the loop at compile time and therefore dynamic loops are not possible on this hardware without performing loops with successive rendering passes. This can drastically affect performance and the computational expense of an algorithm due to requiring additional buffers.

Conditional expressions on older hardware are computed with condition code registers. The outcome of an operation additionally updates a condition code for examination by subsequent instructions. This implies that all instructions must be executed and large bodies of instructions cannot be skipped.

Instruction	Cost (Cycles)
If/EndIf	4
If/Else/EndIf	6
Call	2
Ret	2
Loop/EndLoop	4

Table 3.4: GPU Branching Costs

Recent hardware has included dynamic branching in the fragment processing stage of the pipeline. Kilgariff and Fernando [KF05] lists the expense of branch instructions (see Table 3.4) and it is clear that a substantial overhead is required to successfully include dynamic algorithms. In comparison the same architecture can perform a four component multiply-add and four component dot product in one cycle. In situations where the complete conditional instruction block can be executed without a branch with less cycles than the overhead, condition code registers should be used as a substitute.

3.2 GPU Volume Rendering Algorithms

There are two main approaches to volume rendering on the GPU, object-order and image-order. Object-order approaches are analogous to splatting [Wes90], however exhibit improved coherence between object samples and final image pixels. There is a one-to-one

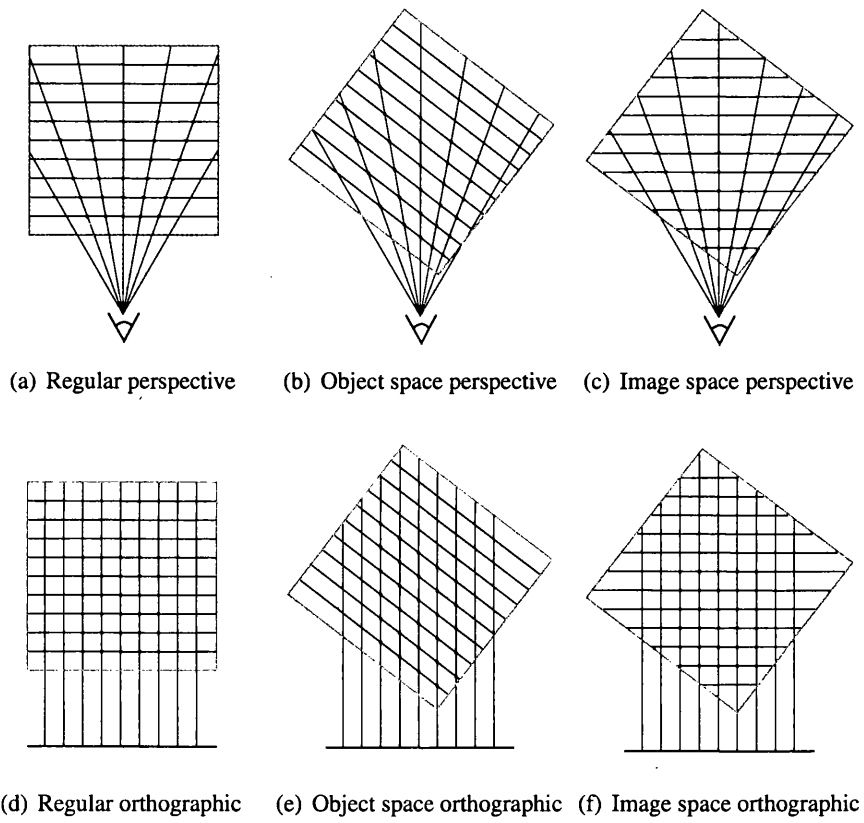


Figure 3.6: Differing proxy slice strategies

mapping from object space samples to the final image plane pixels which highlights a special case of the original splatting algorithm where no footprint computation for multiple final image pixels is required. Since no volumetric primitives can be processed with GPU's, surface primitives have to be employed to encode volumetric approaches. Available primitives include points, lines and planar polygons. The surface primitives employed to encode volumetric approaches are referred to as proxy geometry. Axis aligned proxy slices (or object space proxy slices, see Figure 3.7(b)) and view aligned proxy slices (image space proxy slices, see Figure 3.7(a)) can be used to encode object-order approaches. Image space proxy geometry can also be used to encode image-order ray casting.

Object space sample planes describe proxy geometry that is parallel to a volumes bounding face. Sampling distances encoded in this manner are inconsistent along different rays when using perspective projections (see Figure 3.6(a) and 3.6(b)). Rotation of the volume changes the sampling distances along different rays. Orthographic projections also exhibit differing step sizes along different rays (see Figure 3.6(d) and 3.6(e)), these step sizes also change under rotation of the volume. Rotation of object space sample planes leads to visible gaps through the volume since a proxy slice is infinitely thin when viewed perpendicular to its plane. This effect is eliminated by switching proxy geometries that are most parallel to the image plane. A total of six cases are required, one for each volume face plane (see Figure 3.7(b)). Texture coordinates remain constant for each case.

Image space sample planes are fixed planes parallel to the image plane. Translation operations are not applied to the proxy geometry as it remains fixed parallel to the image plane at all times. Translation operations are applied to the texture co-ordinates for each corner of a proxy geometry. Step sizes using this method also change along different rays when using perspective projection (see Figure 3.6(a) and 3.6(c)) however remain consistent for any rotation of the volume. Orthographic projections exhibit a consistent sample distance with orthographic projection for any rotation of the volume (see Figure 3.6(d) and 3.6(f)).

3.2.1 Object-order Proxy Slice Rendering Using Volume Textures

Most modern GPU's contain hardware to address 3D texture maps with trilinear interpolation. Earlier high-end graphics workstations also exhibited 3D texture mapping hardware capable of trilinear interpolation. Approaches that involve 3D texture addressing can use image or object space proxy geometry. The majority of approaches in the literature are based on high-end graphics workstations because GPU's did not encompass 3D texture addressing hardware.

Wilson *et al.* [WVW94] encode texture co-ordinates in image space sampling planes to allow arbitrary rotations of the volume. This approach was described with high-end graphics hardware. Encoding a unit bounding box will clip parts of a volume under some rotations (where the volume is oriented by 45° for example). The texture co-ordinates are set outside the bounding box of the volume to allow such rotation angles, this results in unused rasterized fragments. 3D texture mapping hardware is utilised to encode direct volume rendering using pre-classification to colour the volume before processing. Hardware resident trilinear interpolation is also utilised to perform fast signal reconstruction. This provides interactive volume rendering by utilising fast hardware operations however no shading is computed.

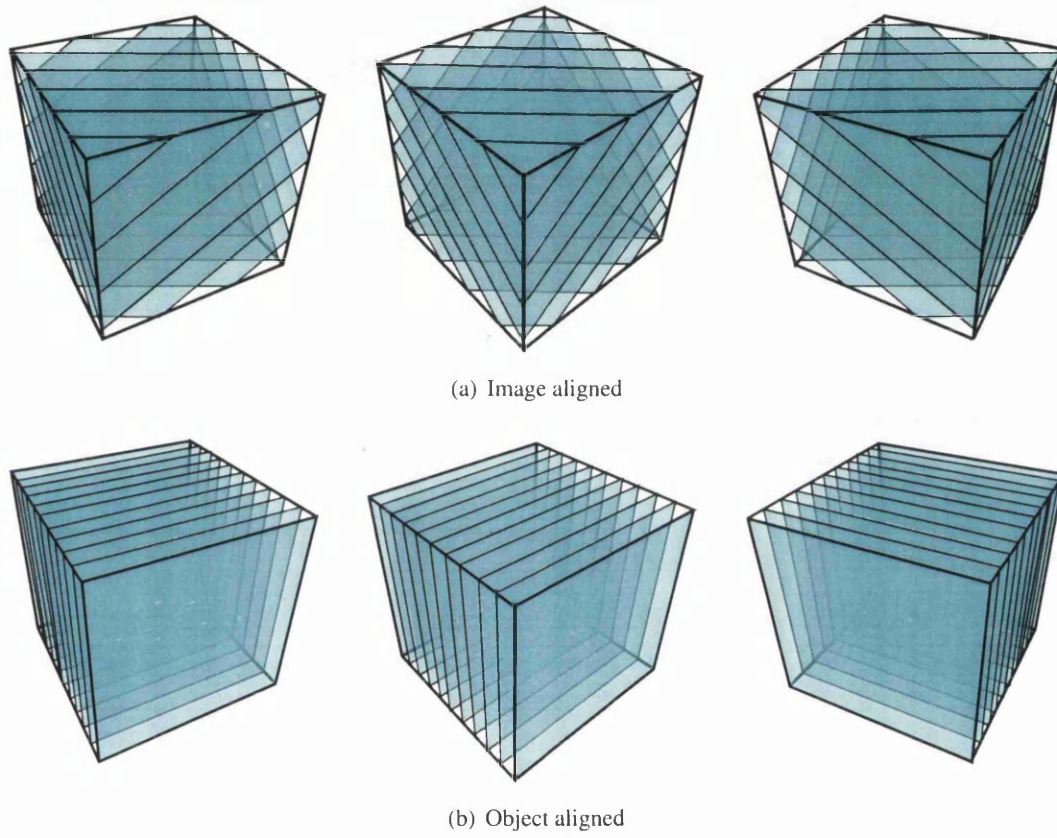


Figure 3.7: Image and object aligned proxy geometries

Cullip and Neumann [CN94] describe a similar image space sample plane strategy although additionally include six clipping planes in order to process the minimum number of fragments defining the interior of the rotated volume bounding box. The cut planes are the unit volume bounding box and are moved with each rotation of the texture co-ordinates. Direct volume rendering with 3D texture mapping hardware on high-end workstations is described. They demonstrate two methods of generating proxy geometry, object space sample planes and image space sample planes for perspective projection. Shading is described with pre-computed shading coefficients based on a voxels gradient. The shading extension does not provide interactive results.

Cabral *et al.* [CCF94] describe 3D texture map direct volume rendering for accelerating numerical radon transforms on high-end workstations. They utilise texture hardware to construct volumes from CT scanner data and render volumes with the standard approach. Image space sampling planes are used for rendering and no shading is computed. Post-classification is implemented using another texture map during fragment processing, the scalar value encountered from the volume is looked up in a transfer function texture map. Interactive rates are reported for this method including post-classification.

Van Gelder and Kim [GK96] introduce shading to 3D texture map direct volume rendering on high-end workstations. A pre-classification scheme is employed to include shading. Gradient magnitudes are used in a pre-processing step to segment the volume into lit voxels and ambient voxels. Each voxel is assigned a gradient index into a quantised gradient lookup table. A lookup table is constructed to map each lit voxel value and gradient index to a $\langle r, g, b, \alpha \rangle$ colour. The size of this lookup table is a function of quantized gradients and segmented gradient magnitudes. A lookup table is also generated for ambient voxels that is independent of gradient, the size of this table is a function of density values for ambient voxels. The volume is then pre-classified according to the lookup tables. This requires processing the entire volume in software and can be time consuming in respect of the volume's size. The volume is then loaded into texture memory of the graphics hardware and rendered using 3D texture map direct volume rendering. If the lighting position or volume orientation is changed, the lookup tables must be recomputed and the volume must be subjected to further pre-classification with these lookup tables. The method detailed does not produce interactive frame rates.

Dachille *et al.* [DKC⁺98] introduce shading at interactive rates by utilizing graphics hardware in the software volume rendering pipeline. Graphics hardware is utilized to perform fast signal reconstruction for ray samples. Proxy geometries are oriented perpendicular to the image plane to reconstruct samples contributing to a row of pixels in the final image. The sample locations are copied from the frame buffer into main memory to perform software ray casting. A lookup table is used for pre-computed Phong illumination. The final image is copied to the frame buffer for display. This method is similar to the shear-warp algorithm [LL94], however since it is mostly image-order, avoids both the shear and warp.

Westermann and Ertl [WE98b] extend 3D texture based direct volume rendering to include shaded iso-surfaces, arbitrary clip geometries and rendering of unstructured grids through tetrahedral projection. Their implementations are described for high-end graphics workstations. Arbitrary clipping geometries are constructed from triangular meshes and rendered to update the stencil buffer. Fragments are subject to a stencil test to determine clipping

locations during rendering. Lit iso-surface rendering is based on the 3D texture map direct volume rendering algorithm by including an alpha test before rendering the fragment into the frame buffer (see Figure 3.2). Additionally if clipping geometry is being considered, the stencil test is also performed against the stencil buffer. A user defined alpha threshold is used to reject transparent and semi-transparent alpha values. Iso-surface shading is approximated using the gradient normal and material density by encoding the gradient normals in the texture maps $\langle r, g, b \rangle$ channels in the $[0, 1]$ range which are rescaled during rendering. The material value is encoded in the texture maps α channel. Standard volume rendering is carried out on the volume and composited gradient normals and material values are rendered into the frame buffer which are subject to a transformation by the colour matrix to compute shading. The application of the colour matrix requires copying of the frame buffer to itself in order to allow pixels to be subjected to frame buffer arithmetic. Ambient and diffuse lighting is defined for parallel light sources positioned at infinity. These methods work on raw volumetric data and provide binary segmentation for iso-surfaces based on the α values, thus only monochrome images are possible due to storing gradient normals in the volume textures colour channels and additionally the matrix multiplication can only consider one light source.

Meißner *et al.* [MHS99] introduce post-classification and shading of semi-transparent materials into the 3D texture map direct volume rendering pipeline for high-end graphics workstations. Unclassified volume data is rendered with the standard 3D texture map direct volume rendering approach. The gradient normals are present in the volume texture map as $\langle r, g, b \rangle$ colour channels. The voxel density is encoded in the α channel of the volume texture map. Gradients have to be stored in the $[0, 1]$ range and are scaled and biased with OpenGL fragment operations. Volume rendering is performed on the proxy geometry and a pixel copy operation is carried out on the resulting frame buffer to invoke colour matrix arithmetic. A colour matrix is defined to calculate the shading intensity which depends upon the voxel's gradient and retains the original density value. The resultant value is used to fetch a classified and shaded colour from a texture map. More advanced classification approaches that use additional information other than a voxel's density are also described. This method allows computation of post-classified semi-transparent direct volume rendering.

LaMar *et al.* [LHJ99] introduce shell rendering to provide consistent sample locations in orthographic and perspective volume rendering using high-end graphics workstation hardware. The premise of the algorithm is to use spherical shell proxy geometry instead of parallel slice geometry. Figure 3.8 outlines the proxy geometry as applied to projection rendering. The proxy geometries remain fixed parallel to the image plane and texture coordinates are transformed to allow rotation of the bounding box. They further describe adaptive multi-resolution rendering of large datasets by subdividing the volume with an octree and rendering octree nodes.

3.2.2 Object-order Proxy Slice Rendering Using 2D Textured Slices

Volume rendering on GPU's that do not have 3D texture mapping capability (2^{nd} generation or earlier cards, see Table 3.3) is approximated using 2D texture maps and substituting tri-

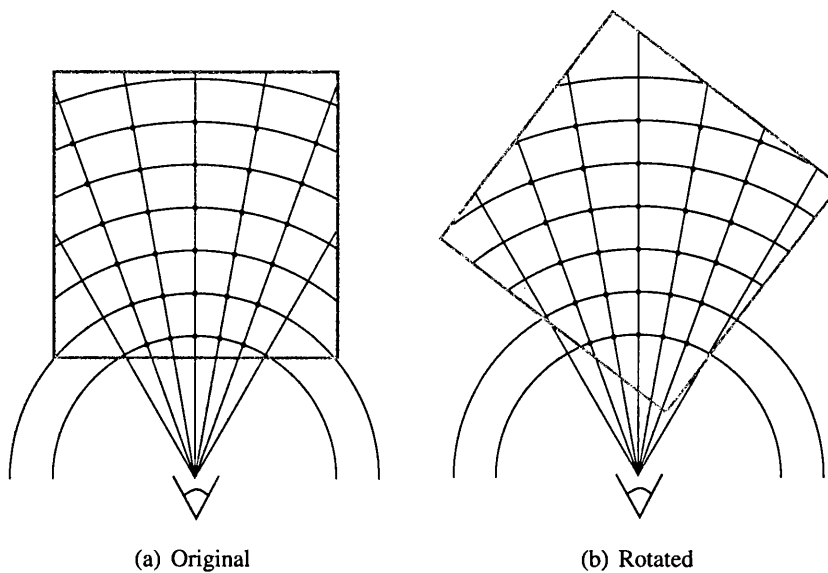


Figure 3.8: *Spherical shell proxy geometry*

linear interpolation with bilinear interpolation. Proxy geometries are employed to represent virtual 2D slices through the volume and are axis aligned (see Figure 3.7(b)). These proxy slices are textured from a set of 2D textures encoding each volume slice's scalar field. Rotation of the proxy geometry produces gaps in the final image due to fragments not being present between slices for a particular axis. To overcome this problem, sets of axis aligned proxy geometries are used for each axis of the volume dataset. To correctly texture each set of axis aligned proxy geometry additional 2D textures are needed for each axis.

2D Texturing techniques do benefit from better memory alignment since 2D texture maps are accessed across a proxy slice, bilinear interpolation is also faster to compute than the trilinear interpolation required for 3D texture maps. This substitution is performed by sampling each 2D texture map on cell faces which avoids the additional interpolation step. In addition this method also benefits the visualisation of large datasets since uploading 2D textures proves less expensive. A dataset that does not fit in GPU memory can be uploaded as a stream of textures during rendering.

Brady *et al.* [BJNN97] introduce a direct volume rendering algorithm for interactive volume navigation based on 2D texture mapping hardware. At the core of this approach to interactive volume navigation with sub-volumes is a 2D texture hardware proxy slice renderer. 2D Axis aligned proxy geometry is textured and blended into the frame buffer to provide a volumetric view. Reported rendering speed of 2D texturing opposed to 3D texturing implementations is greater since memory alignment issues are improved by computing bilinear interpolation within 2D textures. This speed-up is similar to the shear-warp rendering method [LL94] by substituting full trilinear interpolation with bilinear interpolation.

Rezk-Salama *et al.* [RSEB⁺00] describe direct volume rendering on graphics hardware using 2D texture mapping hardware. Object space proxy slices (figure 3.7(b)) are rasterized and textured with a set of 2D textures encoding voxel information. They describe three sets

of proxy geometry and texture sets to overcome visible gaps between proxy geometries when the volume has rotations applied to it. This reduced number of proxy sets is possible by either using the front or back faces of the proxy geometry during rasterization. They further employ multi-texturing to allow more than one sample per proxy slice fragment. This approach allows an interpolation between adjacent virtual proxy slices to perform full trilinear interpolation. These intermediate slices are generated in respect of perspective projection.

Multi-texturing is also explored as a means to accelerate rendering by reducing the amount of proxy slices and encoding multiple steps in each proxy slice. Lit iso-surface rendering can be computed in one pass by using the fragment processor configuration to perform the lighting equation for each sample. The volume texture map contains each voxels gradient normal in the $\langle r, g, b \rangle$ components and the voxels scalar is contained in the α channel. The alpha test is used to reject fragments that do not contribute to the iso-surface by a comparison with the volume scalar in the alpha channel and a predefined iso-value. Semi-transparent rendering is demonstrated with fixed ambient contributions for each voxel by blending the proxy geometry instead of utilising the alpha test to reject fragments. The hardware described only allows two texturing units at a time to be used and post-classification is not described because both texture units are used to provide trilinear interpolation. Post-classification can be implemented at the cost of trilinear-linear interpolation. More recent hardware allows this technique to compute full trilinear interpolation for post-classified images.

3.2.3 Ray Casting

Levoy [Lev88] originally described image-order ray-casting in software. The implementations described in this section describe hardware acceleration techniques for the original ray-casting algorithm.

Westermann and Sevenich [WS01] utilise 2D texturing hardware to accelerate volume ray-casting in software. They employ object space proxy slices and 2D texturing hardware. Standard volume rendering is performed and the depth buffer updated. The alpha test is used to reject fragments from updating the colour and depth buffer. Rendering a frame will yield a view aligned depth buffer representing iso-surface sample locations. Additional slices are added to the proxy geometry set depending on the volumes rotation. This allows the sample distances to be maintained under rotation for orthographic projection (see Figure 3.6(d) and 3.6(e) for uncorrected sample distances). Ray-casting in software is then performed by using depth information obtained from classified volume rendering with proxy geometry.

Krüger and Westermann [KW03] introduce direct volume rendering via ray-casting on the GPU. This approach computes image-order ray-casting entirely on the graphics hardware using multiple passes. This approach allows empty space leaping, adaptive step sizes and early ray termination. In the first pass, the volume bounding box is rendered to avoid processing fragments outside the volume. The second pass processes the back faces of the volume bounding box. These values are used to derive the ray direction vector. These values are rendered into two floating point buffers which are used as texture maps in subsequent passes. This allows computation of orthographic and perspective projections. A

number of passes are then made using the front face proxy geometry to encode ray loops. A ping-pong scheme is used to allow blending since reading and writing to the same buffer is undefined. One buffer will be used as a texture map in a pass for texture access, the other will be a render buffer. The roles of these two buffers are swapped after each pass. Each pass renders proxy geometry (front face of the volume bounding box) to allow access to the starting ray positions in the bounding box. The number of steps and step size is used in conjunction with the ray start location and ray direction vector to derive each passes sample point inside the bounding box. This allows each iteration along all rays to be processed in parallel.

Signal reconstruction, post-classification and lighting is then computed in one pass for each ray step. The result of this pass is written into a buffer to collate the composited ray contributions. Another intermediate pass is performed for each proxy slice that computes early ray termination. If an iso-surface is discovered or the opacity written into the collation buffer is full, the depth buffer is updated so processing of fragments contributing to this ray are halted. This intermediate pass allows the early z test to be utilised. Empty space skipping is also described by using an octree encoding scheme for an additional 3D texture map. One level in a min-max octree is defined to encode a collection of leaf level voxels. This implementation describes encoding an eight voxel neighbourhood at each texel. Multiple samples are described for each pass to limit the number of passes if multiple texture lookups are allowed. Lighting is defined for iso-surface rendering by means of looking up gradient normals from the 3D volume dataset. Lighting must be performed for every fragment processed since no conditional operators are defined. This method requires that the number of passes is known before hand as no stopping criterion is implemented to decide if fragments are still being processed.

Roettger *et al.* [RGW⁺03] implement clipping and oversampling in GPU hardware with pre-integrated rendering (see section 3.3.3). Their approach is to use multiple passes through the graphics hardware to compute an image-order ray casting algorithm with multiple render targets. Early ray termination is described with adaptive step sizes and empty space leaping. The first pass computes the ray sampling locations in a floating point buffer (ray buffer). Ray directions in this approach are computed on the fly. A ping-pong method is then employed to traverse the rays by encoding a ray loop. Ray samples are computed by looking up texturing co-ordinates or ray sample locations from the ray buffer. These samples are then classified with the pre-integration scheme and lit in the first intermediate rendering pass. This intermediate set of ray samples is then blended into another floating point buffer and ray sampling positions are updated according to the space leaping strategy into the ray buffer using multiple rendering targets. The oversampling is described to use 4 volume classification steps instead of one, so 4 pre-integrated samples are composited before blending into the image buffer. The second intermediate pass computes ray termination by updating the z buffer. If the ray leaves the volumes bounding box or the accumulated opacity is approximately full then the depth buffer is updated to reject further fragments. This method can also exploit the early z test. Additionally an occlusion test is used to determine if any fragments were written during the first intermediate pass, each remaining pass is not performed if this criterion is met. The floating point image buffer is finally displayed in the frame buffer.

Miller and Jones [MJ05] and Stegmaier *et al.* [SSKE05] independently introduce single pass ray-casting on GPU's. Using 5th generation GPU's allows the inclusion of branch instructions in fragment shaders. Proxy geometry is not used in these approaches, instead a single image aligned quadrilateral is rasterized into fragments and the entire volume rendering algorithm is computed in the fragment shader. This is analogous to the above approach, however multiple passes and intermediate ping-pong buffers are not necessary. A front-to-back rendering strategy is employed to allow early ray termination, empty space skipping and adaptive step sizes. Sampling distances are better defined in this manner, as parallel proxy geometries are not used to encode one of the spatial directions. This reduces to shell rendering (see Figure 3.8) by computing each step in the fragment shader. This is also possible using multiple pass approaches. Orthographic and perspective projections can therefore be computed with correct sampling positions being maintained. Stegmaier *et al.* [SSKE05] compute ray starting locations and directions in an initial pass. Miller and Jones [MJ05] compute the ray starting locations with texture-coordinates that are passed into the graphics hardware for orthographic projection which removes the first pass. The fragment shader will loop through each ray's sampling positions for each fragment of the rasterized quadrilateral. At each iteration volume rendering can be computed by firstly looking up a trilinear interpolated scalar sample from a 3D volume texture. Subsequently a classification can be carried out with a dependent texture read. These values can then be composited with the standard volume rendering over operator (see Eqn 2.19). This composited structure can be maintained in temporary registers during the loops execution. Since blending into the frame buffer is not computed, higher accuracy blending can be performed in the fragment shader for each sample location of the ray. This is achieved in multi-pass techniques by using high precision floating point render targets, however blending per pass is still computed and the graphics hardware blending units currently do not support full precision. Stegmaier *et al.* [SSKE05] use two nested loops to allow more than 256 ray steps to be computed. Without nesting two loops the hardware describes a maximum of 256 iterations per loop instruction. Miller and Jones [MJ05] only use one loop with multiple samples defined inside the loop. This allows the restriction of 256 ray steps to be removed and additionally benefits from less loop instructions to be considered which are currently costly to perform.

The ray setup described above can be achieved by using texture-coordinates only for orthographic projection. Texture co-ordinates can be trivially set to determine entry and exit points for the volumes bounding box. Using these positions a vector can be created to define the direction of the ray. Step sizes can thus be generated that allow arbitrary sampling of the volume and additionally encode consistent step sizes for arbitrary rotations of the volume. Perspective projections can be generated using this approach, however the linear interpolation scheme used during rasterization will result in sample locations located on proxy slices analogous to Figure 3.6(c). The computation of steps on the fly allows the discontinuities to be removed between perspective ray steps.

3.2.4 Distance Field Rendering

Distance fields describe iso-surfaces and therefore previous iso-surface rendering techniques can be employed. Distance fields generally contain one iso-surface and each voxel describes the distance to the closest point on the iso-surface (see Eqn 2.24). To render an

iso-surface, a binary decision is made on the distance value at the sample location. Generally negative values are used for voxels that are outside the iso-surface, zero for the iso-surface and positive values for the interior of an object. Older generations of GPU's do not contain floating point texturing units so a quantisation is required to map the distance values into an available range (see Eqn 3.1). Typically older generations were capable of 8 or 16 bit unsigned integers which are treated as fixed point scalars internally in the $[0, 1]$ range. This pre-processed quantisation of the distance field will introduce artifacts due to under-sampling the original data resolution and also introduce a costly pre-processing step. Additionally a remapping might be applied during fragment processing to compute over the original range if required (see Eqn 3.2).

Yamazaki *et al.* [YKI03] perform distance field rendering on GPU's by utilising a 3D texture hardware volume rendering approach for iso-surfaces. Pre-integration is discussed to accurately represent iso-surfaces between proxy-geometry slices and an interpolation weight transfer function generation method is outlined for multiple regions in the distance field. Generally considering if the iso-surface is present between samples provides much improved image quality by approximating if the original signal contained the iso-surface between points.

Image-order ray casting algorithms can be adapted to accelerate distance field rendering by skipping the distance defined at a voxel along the ray if the iso-surface is not encountered. Generally floating point texture targets are required for this approach since quantised distance values would have to be remapped for each sample and the quantisation might not accurately represent the correct distance. 5th generation GPU's are capable of reading and writing arbitrary floating point textures and buffers, where older generations must quantise to a fixed point representation and a possible remapping during fragment processing. The ability to use floating point textures cuts out the quantising pre-processing step and possible multiplications during fragment processing for remapping the values.

A single pass image-order ray caster on the GPU can be defined to include early ray termination, empty space skipping and adaptive step size sampling. These methods can also incorporate consideration for iso-surface values lying between sample positions such as pre-integrated rendering techniques. The single pass renderer allows floating point texture targets as well as increased floating point precision during fragment processing due to improvements in hardware architecture. Interactively segmenting the distance field provides a faster mechanism for traversing the volume on this hardware because individual samples can be skipped and rejected.

Figure 3.9 shows an example ray being sampled at regular intervals. The first lookup at sample *a* produces the encoded distance to the object as represented by the line from the sample position to the closest point on the surface. This distance may be skipped along the ray since it is guaranteed that it is at least this distance to the surface. The next sample location *b* does not contain the surface but three samples along the ray have been skipped upon sampling this scalar.

During computation of space leaping the negative and positive values must be taken into account to accurately perform a step along the ray. In simple iso-surfacing situations this is not necessary as once the iso-surface is intersected, early ray termination can be performed.

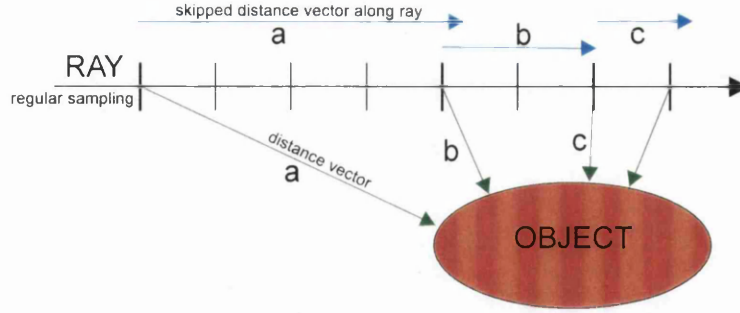


Figure 3.9: Distance field rendering: Empty space leaping along a ray. Samples are taken at the bold intervals along the ray where the distance vector allows stepping over a number of samples. The distance is rounded down to the equidistant sample points.

Extended iso-surfacing techniques that choose iso-values outside of the originally defined surface in the distance field require consideration of the sign. In these cases, the sign can be removed by remapping the entire distance field to encode 0 as the maximal value, with each other value having the maximal value subtracted from it. These values can then be made positive, to describe a distance value in respect of the maximum. This allows arbitrary iso-values to be correctly computed as the computation skips along the distance from the sample location (remapped) and the iso-value (also remapped). The equations outlined below assume this linear mapping. Without this mapping internal samples for the object will produce negative increments along the ray. If this mapping is not carried out explicitly, an *abs* function can be used to avoid this problem.

Space leaping can be carried out using two approaches, both of which perform adaptive step sizes for sampling close to the object's surface. The correspondence between sample distance and voxel width is required in the first approach. Each distance field scalar represents the distance in voxels to the closest point on the iso-surface, thus only the vector describing the distance to skip one voxel is required to perform an advancement along the ray for a distance value above 1. When the distance values encountered are a voxel or less, sampling must be performed using a constant step vector to ensure that the surface is accurately found efficiently. This is due to sub voxel precision possibly requiring many small advancements when the distance value is under the one voxel length threshold. Since the step vector can be an arbitrary length, the step vector size can be larger than a defined skipping vector for values over one voxel. Therefore the largest vector is chosen to advance the ray position (see Eqn 3.3).

$$\text{step}(d, s, v, i, f) = \begin{cases} (v(f - i) - 1.0)d & \text{if } (f - i) - 1.0 > 1.0 \\ & \text{and } v(f - i) - 1.0 > s \\ ds & \text{otherwise} \end{cases} \quad (3.3)$$

where $d \in \mathbb{R}^3$ is the normalised direction vector, $s \in \mathbb{R}$ is the step size, $v \in \mathbb{R}$ is the voxel step size, $i \in \mathbb{R}$ is the iso-value and $f \in \mathbb{R}$ is the sample encountered along the ray. The remapping described above is assumed.

The second method disregards the acceleration of choosing the longest vector to skip along

a ray and only allows the step vector to be employed when the sampled distance values are below a given threshold. Generally a good threshold is one voxel, although a larger adaptive region is sometimes used to ensure correct sampling is achieved with possible small irregularities in the distance field. This method is not as efficient with large step vectors, however produce a comparable result when the step vector is smaller (see Eqn 3.4). These adaptive rendering strategies with empty space leaping can be regarded as refining the ray marching process towards the object surface Figure 3.10.

$$step(d, s, v, i, f) = \begin{cases} (v(f - i) - 1.0)d & \text{if } (f - i) - 1.0 > 1.0 \\ ds & \text{otherwise} \end{cases} \quad (3.4)$$

where $d \in \mathbb{R}^3$ is the normalised direction vector, $s \in \mathbb{R}$ is the step size, $v \in \mathbb{R}$ is the voxel step size, $i \in \mathbb{R}$ is the iso-value and $f \in \mathbb{R}$ is the sample encountered along the ray. The remapping described above is assumed.

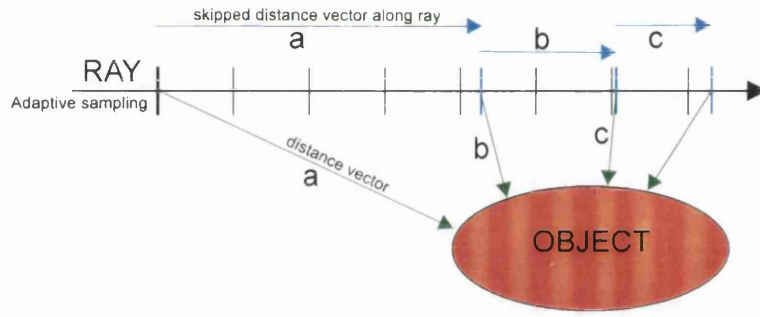


Figure 3.10: Distance field rendering: Adaptively rendering along a ray. Samples are taken at bold intervals along the ray. These samples are not rounded down toward the equidistant samples and are taken at exactly the distance vectors length along the ray.

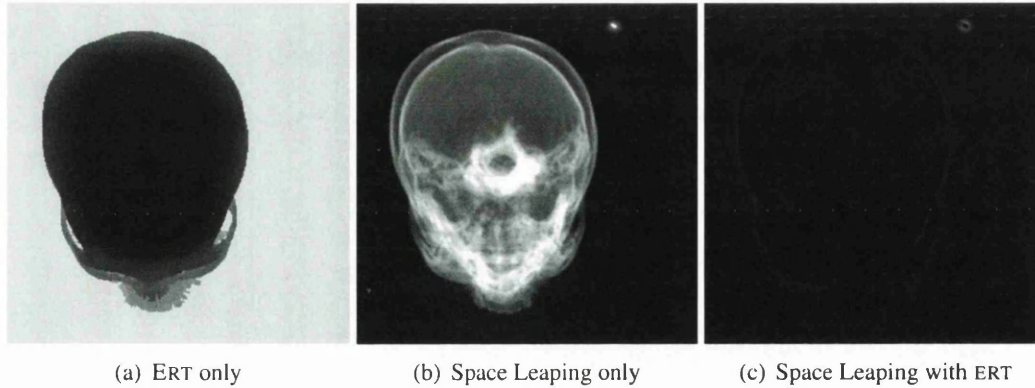


Figure 3.11: Distance field ray samples through volume with; (a) early ray termination, (b) empty space leaping and (c) empty space leaping with early ray termination. The grey scales define the number of samples taken along the ray with black being no samples towards white being the maximum samples.

Figure 3.11 shows an example rendering of the *CTHeadDist* dataset with empty space leap-

ing employed as an acceleration method. The number of samples evaluated along a ray are encoded as monochrome values where black represents no samples and white represents 256 samples. The difference in evaluated samples along a ray between early ray termination, empty space leaping and empty space leaping with early ray termination are given. It is clear that the latter method is the most efficient in terms of visited samples along a ray. A trade off generally exists around performing the space leaping function and producing less instructions to consider by sampling at regular intervals. Most real-world examples benefit from this approach since it reduces the burden of sampling many times along the ray where no contribution to the final image is present. The *CTHead* for example contains regions of interest (skin and bone) in the centre of the dataset. Defining a distance field for this dataset (e.g. *CTHeadDist* for bone) allows skipping upto the bone structures, sampling the bone structures and skipping empty space inside the object if further sampling is required. A more scalable solution to the volume rendering problem is thus defined with empty space leaping with early ray termination.

This function can be pre-computed as a quantised 1D lookup table. Since linear interpolation is resident for texture fetches on GPU architectures this quantised approximation is acceptable since the function is linear in nature. A mapping of all voxel values must firstly be undertaken as a pre-processing step (see section 3.1.6) since a dependant texturing step for each voxel's scalar distance value encountered along the ray must be used to address the lookup table space which is defined in the $[0, 1]$ range. This benefits older generation GPU's that are not capable of addressing outside this range, and additionally use texture maps that contain values outside the $[0, 1]$ range since a remapping during shading is not required.

The acceleration technique of empty space leaping along a ray is most suited to image-order techniques since the front-to-back or back-to-front ray casting can benefit directly. Object-order approaches do not benefit from this approach. It is therefore possible to compute a distance field for a given volume dataset exhibiting scalar densities to accelerate it's rendering. This requires more GPU memory as two volume textures are required if the gradient normals are stored in the target dataset.

3.3 Improvements

This section explores developments made to the standard GPU volume rendering framework. Section 3.3.1 reviews clipping geometries to the GPU volume rendering pipeline to enable cut away views of the volume dataset. Section 3.3.2 covers methods to compute higher order signal reconstruction filters than are available as hardware operations. Section 3.3.3 explores pre-integrated rendering techniques to consider an additional sample at each location.

3.3.1 Clipping

Westermann and Ertl [WE98b] introduced arbitrary clip geometries using the stencil buffer and stencil test available in the GPU hardware pipeline. The stencil buffer must be updated for each proxy slice to be correctly clipped. The geometry is a tessellated triangular mesh.

A proxy slice is then subjected to fragment processing, blending is then performed into the frame buffer after an intermediate stencil test for the proxy slice against the stencil buffer.

Weiskopf *et al.* [WEE02] describe arbitrary clipping geometries for GPU volume rendering. Two algorithms are presented to clip volumes during proxy slice based rendering. The first method utilizes depth information from clipping geometry rendered into the depth buffer to perform clipping. This method can accommodate two clipping boundaries by rendering the clipping geometry front faces into a texture map and also rendering the back faces of the clipping geometry into the depth buffer in a different pass. During rendering passes the fragments depth is shifted with the results of the first stages texture map. This leaves depth values in front of the viewing frustum and thus these fragments are clipped. The depth buffer test enables removal of fragments behind the clip geometry using the depth test. The second approach is to use a per fragment kill operation in fragment processing by addressing a binary volume texture representing the clipping geometry. This approach does not use the depth buffer which enables its use as a further speedup mechanism.

Weiskopf *et al.* [WEE03] expand the clipping approaches to include corrected shading. By clipping though the centre of a volume iso-surface, gradients are not well defined for the clipped iso-surface. They employ distance fields as clipping geometries to determine if the iso-surface being rendered is close to the clipping geometry. If the clipping geometry is close to the clipping geometry, the distance fields gradients are used for surface rendering instead of the original gradient. The process of classification must still be carried out firstly as materials discovered close to the clipping geometry might not be part of the iso surface. Roettger *et al.* [RGW⁺03] expand lit clip planes to include pre-integrated classification as well as standard classification.

3.3.2 Signal Reconstruction

Hadwiger *et al.* [HTG01, HVTH02] describe higher order signal reconstruction for GPU hardware. A flexible convolution framework is described which exploits multiple passes and multi-texturing hardware to compute arbitrary filter kernels. The premise of this algorithm is to reverse the filtering process to accommodate graphics hardware. Usually an output sample is generated by convolving over several neighbouring samples. These neighbouring samples will be subject to many output sample positions during *gathering* calculations. Reversing this process leads to progressively solving the filter kernel in a *distributed* manner, each output sample point will distribute to its neighbours.

Multiple passes through the graphics hardware are used for each neighbour of a sample point, the bounds of the filter kernel dictate the number of passes. Each pass applies the filter kernel which is stored in multiple texture maps, each texture maps represents a unit portion of the filter kernel. The results of this work are high quality filtered images that are computed in real time. However this strategy when applied to volume rendering over 3D textures will increase the complexity of standard volume rendering techniques and result in sub real-time display.

3.3.3 Pre-Integrated Classification

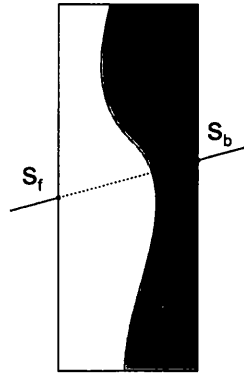


Figure 3.12: Slab encoded between two proxy slices s_f and s_b ; representing the front and back slice. The blue region defines the object boundary that is the subject of reconstruction.

Engel *et al.* [EKE01] introduce pre-integrated volume rendering. This approach attempts to further approximate the volume rendering integral. One key problem with 3D texture based direct volume rendering is that many proxy geometries are needed to accurately reconstruct the signal between adjacent proxy slices. Important information can be missed due to under-sampling the z direction with proxy slice geometries. The addition of more proxy geometry solves the under-sampling of regions between slices, however results in the loss of interactive display in many cases. Pre-integrated volume rendering considers the signal between adjacent slices by performing an integration of the transfer function.

The goal of pre-integrated classification is to minimise sampling artifacts and under-sampling by performing two integrations, one for the scalar field and another for the transfer function. Post-classification is employed and the lookup table is pre-computed according to a 1D transfer function lookup table of scalar values to $\langle r, g, b, \alpha \rangle$ colour values. This method does not treat each proxy slice as an infinitely thin plane through the volume, instead it encodes each proxy slice as a slab (see Figure 3.12). This is calculated by altering the standard fragment processing technique of post-classification to include two lookups into the volume texture map.

The first lookup fetches the trilinear interpolated scalar value for a sample point analogous to post-classification. The second lookup fetches the trilinear interpolated scalar representing the next sampling position along the ray (the next proxy slices sampling location). A pre-integrated transfer function is pre-computed into a 2D lookup table to be addressed by the front scalar value and back scalar value. This lookup table contains integrated colours derived from the original 1D transfer function and describes an integration between the front and back scalar colours. Additionally step size must be taken into account as alpha correction must be performed, the original description describes a 3D lookup table with step size being the third dimension. In practice it is desirable to compute the 2D lookup table with alpha correction included such that the entire 3D lookup table does not have to be generated for each alteration of the transfer function.

High frequencies in data can be better approximated using pre-integration and additionally

less slices are required to accurately represent a volume without under-sampling of the data. Generation of the pre-integrated transfer functions are described in Equation 3.5. Simplified versions of these equations for computation are given in Equation 3.6.

$$\alpha_i = 1 - \exp \left(- \int_0^1 \tau \left((1 - \omega) s_f + \omega s_b \right) d\omega \right) \quad (3.5)$$

$$c_i = \int_0^1 c \left((1 - \omega) s_f + \omega s_b \right) \exp \left(- \int_0^\omega \tau \left((1 - \omega') s_f + \omega' s_b \right) d\omega' \right) d\omega$$

where s_f and s_b are the values obtained from the transfer function for front and back samples respectively, d is the sampling distance. The colours are associated.

$$\alpha(s_f, s_b, d) = 1 - \exp \left(- \frac{d}{s_b - s_f} (T(s_f) - T(s_b)) \right) \quad (3.6)$$

$$T(s) := \int_0^s \tau(s) ds$$

$$c(s_f, s_b, d) = \frac{d}{s_b - s_f} (K(s_b) - K(s_f))$$

$$K(s) := \int_0^s c(s) ds$$

where s_f and s_b are the values obtained from the transfer function for front and back samples respectively, d is the sampling distance. The colours are associated.

Figure 3.13 shows a 2D pre-integrated transfer function table obtained from a standard 1D post-classification table (see Figure 3.14).

Slab rendering is further extended to iso-surface rendering by generating a 2D transfer functions encoding multiple iso-values for front and back sample scalars. Iso-surfaces are treated as a special case where an interpolation weight is computed to accurately interpolate between front and back positions or gradient normals. The lookup table is computed such that the first iso-surface that is intersected between the front scalar and back scalar is encoded into a 2D lookup table for colour and interpolation weight (see Figure 3.15(b)), thus no integration takes place. The first iso-surface discovered in a slab is considered to occlude all further iso-surfaces since it will be fully opaque. A further channel can also be used to describe semi-transparent multiple iso-surfaces by including an opacity value. In this case only one the first iso-surface discovered in the sample slab will be included in the final image. Additionally two colours can be defined per iso-surface to facilitate front and back face rendering (see Figure 3.15(c)). Generally back face rendering must firstly reverse the gradient to successfully be included in lighting models. Figure 3.16(a) shows a

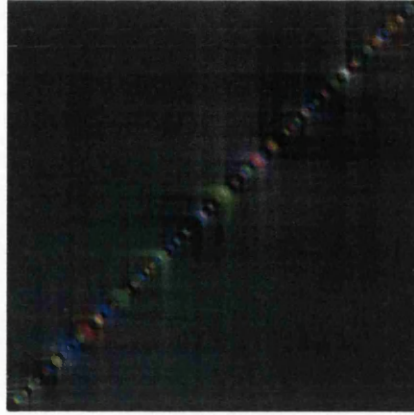


Figure 3.13: Pre-integrated transfer function table, opacity discarded for clarity. The bottom edge represents the front slice value encountered whilst the left edge represents the back slice value encountered during rendering

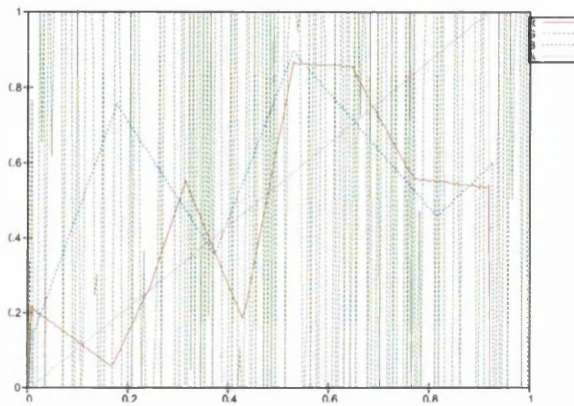


Figure 3.14: 1D Transfer function, R and B are piecewise linear, G is random and A is identity

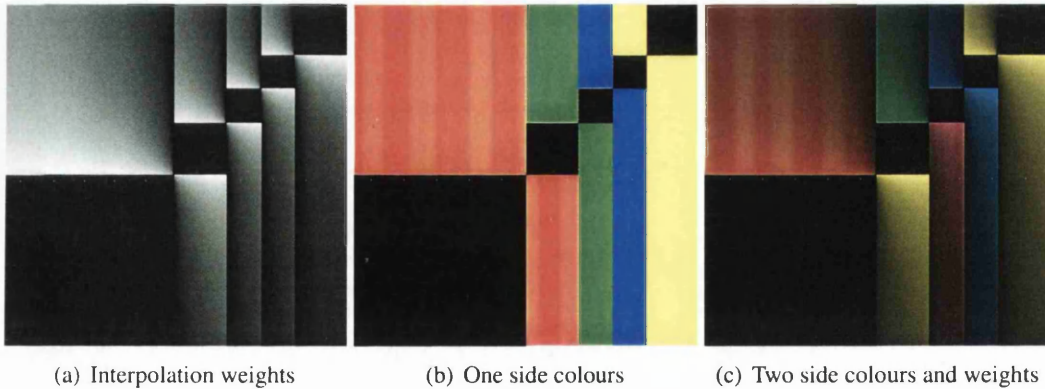


Figure 3.15: Iso-surface lookup tables for slab rendering, iso-surfaces at 0.5, 0.65, 0.75 and 0.85 where (a) is the interpolation weights, (b) is the front face only colours (back face is rendered as front face) and (c) both front and back faces with differing colours with the addition of the interpolation weights. The bottom edge represents the front slab value and the left edge represents the back slice value.

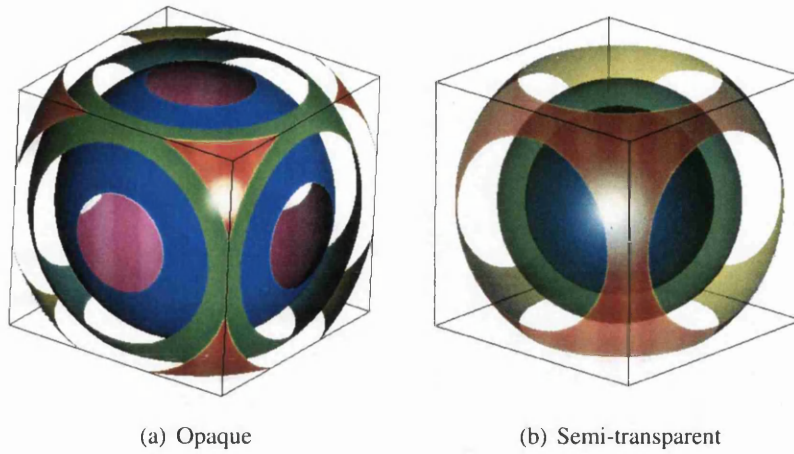


Figure 3.16: Multiple iso-surfaces of the *SphereDist* dataset

front and back face multiple iso-surfacing of the *SphereDist* dataset, figure 3.16(b) shows a semi-transparent variant of the same approach.

The interpolation of gradients between the samples with respect to a pre-calculated interpolation factor allows an accurate gradient to be derived for the exact position of the iso-surface within the slab being considered. The implementation described suggests that another lookup table should be used for interpolation factors, however on the hardware described the maximum amount of textures had been used. This does not allow semi-transparent iso-surfaces since the alpha value is used to store the interpolation weight. The extra table is required since a full $\langle r, g, b \rangle$ triple is required for arbitrary colouring of iso-surfaces and the interpolation factors must be stored. Two methods for including the interpolation factor and the alpha value in the lookup table are described. Both methods remove either alpha detail or colour detail from the lookup table to allow inclusion of the interpolation weights. These methods can produce significant artifacts due to quantising data or removing a channel to a constant value. By using an additional one channel texture map to contain the interpolation weights, semi-transparent iso-surface rendering is possible without compromising image quality.

Lighting is approximated for semi-transparent direct volume rendering with a pre-integration table. Since no iso-value information is available to correctly derive the slabs gradient, a constant blend of front and back gradient are used instead. The lighting is performed in respect to the pre-integrated tables colour and opacity information.

$$weight(s_f, s_b, s_{iso}) = \frac{s_{iso} - s_f}{s_b - s_f} \quad (3.7)$$

Lum *et al.* [LWM04] improve pre-integrated classification to iron out discontinuities in lighting. Previous methods treated iso-surfaces as a special case, with the assumption that the first iso-value discovered in a slab is used to derive the gradient and colour for lighting. This method is not well defined for more than one iso-surface in a slab (see Figure 3.17).

Additionally semi-transparent volume rendering without iso-values is not well defined since gradients are constantly interpolated to the centre of a slab. A weighting for front and back slices is computed with an additional lookup table for diffuse and specular lighting components. These weighted lighting contributions are then interpolated for the slab. This method irons out discontinuity where large changes in gradients are present, and further refines situations where multiple iso-surfaces are present in one slab. Multiple semi-transparent iso-surfaces are not treated as special cases as the interpolated lighting derived from the pre-integration table is sufficient to remove these ambiguities. Whilst this method removes ambiguity of multiple iso-surfaces and improves lighting calculations in semi-transparent volume rendering the extra burden of performing two lighting contributions decreases performance. Additionally since iso-surfaces are not treated as special cases, this method suffers from more artifacts in these cases since the iso-surface position is not defined explicitly in a slab sample.

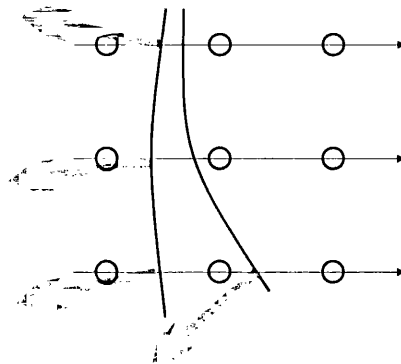


Figure 3.17: *Lighting discontinuity in slab rendering for interpolated gradients. Sample points are represented as circles. A lighting discontinuity can be observed when two surfaces are running through the same slab and one of the surfaces moves between slabs. The normals defined here introduce a lighting artefact.*

They further improve the efficiency in calculating the pre-integrated table with subrange integration. They note that opacity correction for the transfer function must take place to correct differences in the span of values integrated for a sample. An acceleration is derived with corrected opacity by observing that the diagonals of the pre-integration table maintain a fixed span of integrated samples for each sample derived. By calculating on the diagonals of the table, extra values can be composited on the next diagonal of the table which allows re-use of previous results. This improves the run-time of the algorithm over the brute force table generation scheme.

The alpha correction equation for subrange integration accelerated pre-integrated classification table generation. This correction is applied before calculating the pre-integration table:

$$\alpha = 1 - (1 - \alpha_s)^{\frac{d}{|j-i|}} \quad (3.8)$$

where α_s is the opacity contained in the original transfer function, d is the sampling distance and $|j - i|$ is the width of the span of samples to integrate.

3.4 Comparison

Three algorithms are explored in this section for rendering performance and output quality. Further investigation is given to each algorithms respective complexity. Orthographic projections are considered as the basis to compare algorithms without interference due to incorrect sampling frequency and additional overhead to compute perspective ray directions. The term slice sampling is used to describe the sampling of a point in 3D space and slab sampling is used to describe taking two samples and performing an integration or interpolation to sample the space occupied between these sample points. The algorithms described are direct volume rendering:

- Object-order proxy slice rendering (OOP), analogous to splatting with back-to-front sampling and compositing
- Image-order multiple pass ray-casting (IOM) with front-to-back sampling and compositing
- Image-order single pass ray-casting (IOS) with front-to-back sampling and compositing.

Further investigation is given into slice and slab rendering for both fuzzy classification, iso-surfacing and multiple iso-surfacing:

- Fuzzy classification, Slice sampling, Post-classification with 1D lookup tables
- Fuzzy classification, Slab sampling, Pre-integrated classification with 2D lookup tables
- Binary Classification, Slice sampling, Single lit iso-surfaces
- Multiple Iso-surfaces, Slab sampling, multiple lit iso-surfaces. Includes slab sampled single iso-surfaces.

Figure 3.18 provides a screenshot of the software framework used to compare volume rendering algorithm hybrids from the preceding sections. This software is designed around the object oriented paradigm and is written in C++[Str00]. The object oriented class hierarchy allows new GPU rendering techniques by simply overriding a generic rendering class and providing the implementation specific code. Classes are also provided to implement transfer function table generation and volume dataset mapping to 3D textures.

The application framework is written using the OpenGL API[SA] and GLUT[Kil96] a utility windowing framework which are programmable from C [KR88]. A simple C++ wrapper hierarchy has been implemented to allow object oriented programming for GLUT and OpenGL. This implementation is therefore cross-platform since no operating system specific instructions are used. This mechanism allows easy inclusion of OpenGL extensions which are extensively utilised for rendering algorithms.

Classification lookup tables can be pre-computed to include associated colours (see Eqn 2.17) and alpha correction (see Eqn 2.20). Each test conducted includes interactive rotation using a virtual trackball algorithm [HSH04], interactive transfer function change (including

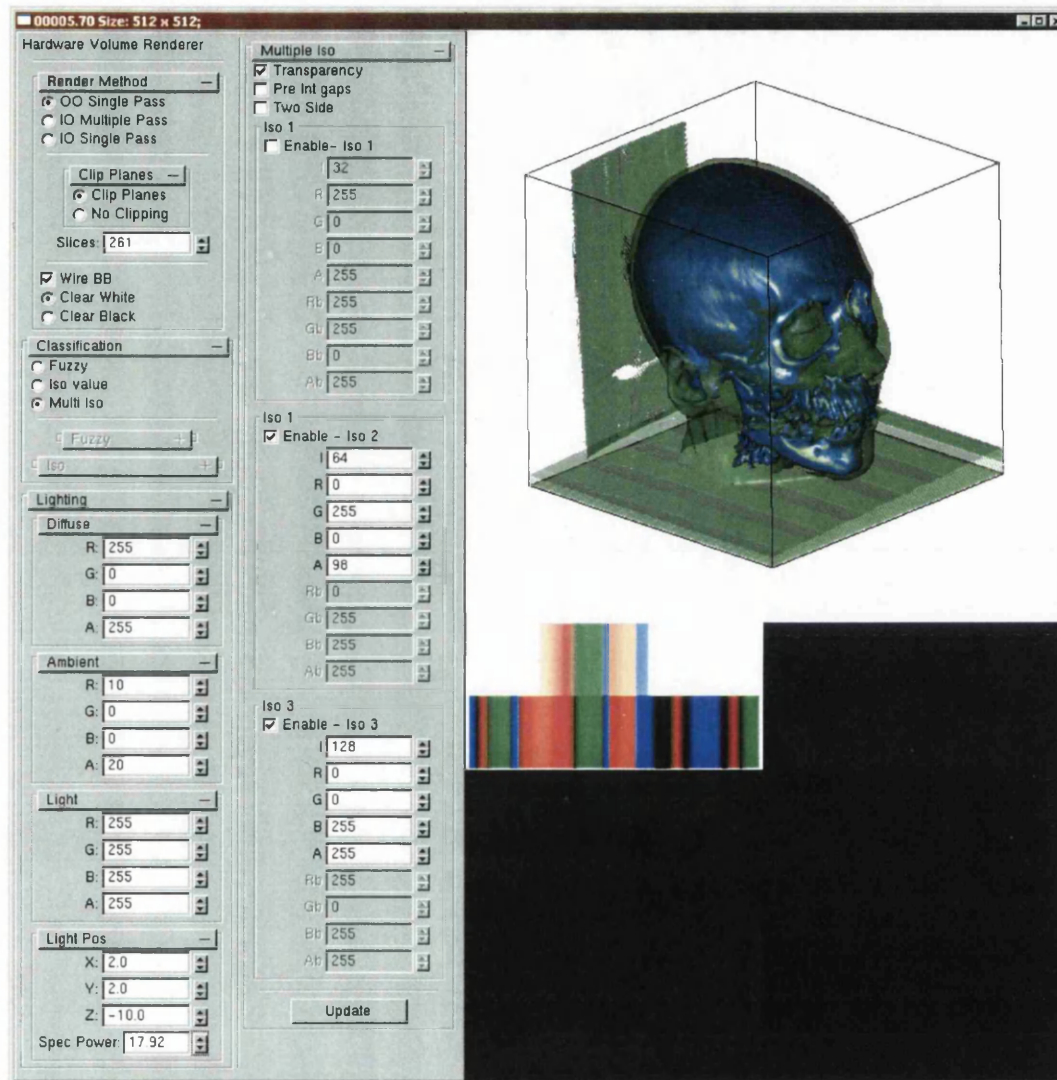


Figure 3.18: Screenshot of software testing environment

pre-integration tables [LWM04]) interactive sample frequency changes, interactive lighting condition changes and iso-value changes where applicable.

	OOP				IOM				IOS			
	Pst	Pre	Iso	Int	Pst	Pre	Iso	Int	Pst	Pre	Iso	Int
ERT	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DS	n/a	n/a	No	No	n/a	n/a	m	m	n/a	n/a	Yes	Yes
ESL	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PS	n/a	No	n/a	No	n/a	m	n/a	m	n/a	0	n/a	0
BP	8	8	n/a	n/a	16	16	n/a	n/a	32	32	n/a	n/a
Buffers	1	1	1	1	6	6	3	3	1	1	1	1
Passes	1	1	1	1	$2\frac{n}{m}$	$2\frac{n}{m}$	$\frac{n}{m}$	$\frac{n}{m}$	1	1	1	1
OP	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PP	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DB	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes
MRT	No	No	No	No	Yes	Yes	Yes	Yes	No	No	No	No
Section	3.4.2		3.4.3		3.4.6		3.4.7		3.4.10		3.4.11	

Table 3.5: Rendering approach and segmentation method comparison matrix. *Pst* is post-classification, *Pre* is pre-integrated classification, *Iso* is single iso-surface rendering, *Int* is interpolated iso-surface rendering or multiple iso-surface rendering. *ERT* is early ray termination, *DS* is deferred shading, *ESL* is empty space leaping, *PS* is previous sample reuse expressed in number of texture lookups in addition to the lowest required for the technique where *m* is the number of samples per pass, *BP* is blending precision in bits per sample, *Buffers* is the base number of off-screen buffers required by the algorithm, *Passes* is the number of passes required to sample the volume (worst case) where *n* is the number of samples along a ray and *m* is the number of samples per pass, *OP* denotes correct sampling for orthographic projection, *PP* denotes correct sampling for perspective projection, *DB* is the necessity of the approach to perform dynamic branches, *MRT* is the requirement to use multiple rendering targets to compute speed-ups.

Gradient computation is treated as a pre-processing step and is not computed on the fly. There is also no gradient filtering applied in order to work directly with pre-computed gradients defined with central differences. Where gradients are used the volume texture is constructed so that the $\langle r, g, b \rangle$ channels contain the gradient normal and the α channel contains the scalar field. Volumes can be defined with 8, 16 or 32 bit precision. Generally these volumes will be defined over the $[0, 1]$ range which requires gradients to be remapped on the fly to the $[-1, 1]$ range. However two simple fragment shader instructions are required to remap the range. Floating point volume textures are capable of defining values outside the $[0, 1]$ range and do not require remapping of the gradient normals. Transfer function textures considered are 8 bit 4 channel lookup tables with 256 locations. These transfer function dimensions are usually adequate for many real world applications including high frequencies. Details of hardware implementation specific issues such as branching and fragment culling are presented where necessary.

A cut down point lighting model is also used and evaluated in the fragment processing stage of the pipeline. A function *lighting*(*norm*, *pos*, *light*) is provided to all fragment shaders that compute this model where *norm* is the eye-space gradient normal vector, *pos* is the

eye-space lighting vector and *light* describes the ambient, diffuse and light colours with the addition of the specular power. This model is an approximation [Cor06] of Blinn-Phong [Bli77] shading (see Eqn 3.9).

$$I = I_a + I_d I_l (N \bullet L) + I_l (N \bullet H)^m \quad (3.9)$$

where I is the resulting colour, I_a is the ambient colour, I_d is the diffuse colour, I_l is the light colour, N is the gradient normal, L is the light vector, H is the half-angle vector and m is the specular power. Specular colour is assumed to be white.

Each rendering approach and classification method is explored in separate sections. Table 3.5 gives an overview of these techniques and describes each approaches characteristics. Some of these characteristics have workarounds which are discussed, however this table represents the abilities of each algorithm as implemented in this work.

3.4.1 OOP Rendering Framework

The algorithm employed for this test is based upon section 3.2.1. Image aligned proxy slices are used to encode volume sampling positions. Clipping planes are used to reduce the burden of fragment processing during rendering. Comparison for no clipping planes is also presented. A 75% increase in total fragments would result with no clipping as the proxy geometry set must be large enough to cope with arbitrary rotations of the volume bounding box. The viewport sized proxy geometry is therefore set 75% larger to account for these rotations. The clip planes are positioned to form a bounding box and clip away redundant fragments that fall outside the $[-1, 1]^3$ range.

```

texStep = 1.75 / slices
vertStep = texStep * 2
curTexFront = 1.375 - texStep
curTexBack = 1.375
curVert = 1.75
for (i = samples; i >= 0; i--)
    beginQuad()
    texCoord1(-0.375, -0.375, curTexFront)
    texCoord2(-0.375, -0.375, curTexBack)
    vertex(-1.75, -1.75, curVert)
    texCoord1(1.375, -0.375, curTexFront)
    texCoord2(1.375, -0.375, curTexBack)
    vertex(1.75, -1.75, curVert)
    texCoord1(1.375, 1.375, curTexFront)
    texCoord2(1.375, 1.375, curTexBack)
    vertex(1.75, 1.75, curVert)
    texCoord1(-0.375, 1.375, curTexFront)
    texCoord2(-0.375, 1.375, curTexBack)
    vertex(-1.75, 1.75, curVert)
    endQuad()
    curTexFront -= texStep
    curTexBack -= texStep
    curVert -= vertStep
endfor

```

Figure 3.19: Image aligned proxy geometry generation


```

fragment vertexShader(vertex, modelViewMatrix, textureMatrix)
    fragment.pos = vertex.pos * modelViewMatrix
    fragment.tex0 = vertex.tex0 * textureMatrix
    fragment.tex1 = vertex.tex1 * textureMatrix
    fragment.tex2 = vertex.pos * textureMatrix.inverseTranspose

```

Figure 3.20: OOP vertex shader

Corresponding texture coordinates are in the $[0, 1]^3$ range and are also oversized by 75%. Each proxy geometry slice is issued 3D texture coordinates to address the volume texture (see Figure 3.19). The order which vertices are defined is important to the API since this determines whether it is front facing or backwards facing. The vertex coordinates are issued in the $[-1.75, 1.75]$ range so that the coordinate system origin is in the centre of the volume bounding box such that translations are not required before rotations.

The texture coordinates are in the $[-0.375, 0.375]$ range to allow access without altering texture coordinates on the fly to the texture blocks in GPU memory. This requires a translation to compute respective rotations around the origin in texture coordinates space. This strategy allows the texture coordinate space to represent volume object space coordinates. Figure 3.19 demonstrates image aligned slice generation for an arbitrary number of samples. This strategy will therefore sample a volume with fewer slices than generated if there is no rotation applied to the volume. The unused slices and fragments are subject to clipping before expensive calculations in the fragment shader, however these additional slices and fragments must be present to allow correct rotations of the volume.

Volume bounding box rotations are accomplished by transforming the texture coordinates alone. The vertex shader (see Figure 3.20) is configured to apply the modelling transformations to the geometry, transform the texture coordinates for volume sampling locations and additionally compute the eye space position for eye space lighting calculations in the fragment shader. After transformation in the vertex shader, the rasterizing hardware will interpolate the texture coordinate sets across the proxy geometry faces for use in the fragment shader.

Fragment shaders for each algorithm will perform volume lookups and dependent texturing for classification. Each fragment's result is blended into the frame buffer if blending is to be computed by using the back-to-front compositing equation (see Eqn 2.18) during the alpha blending stage of the pipeline. Current graphics hardware allows up to 16 bit floating point precision on blending operations per colour channel, however the frame buffer for display is 8 bit fixed point precision per colour channel. Rendering to an off-screen buffer is required to take advantage of 16 bit blending per colour channel, however incurs a performance penalty and requires an additional pass to display the results in the frame buffer.

The discussion is restricted to 8 bit fixed point blending per colour channel into the frame buffer as multiple passes are required for higher precision. Two sets of coordinates are defined for each vertex to allow slab rendering algorithms which require locations to fetch voxels for front and back samples. The standard algorithm does not use the second set of texture coordinates as sampling is restricted to one location. The modelling and projection matrices are unchanged to allow the proxy geometry to remain image aligned at all times including bounding box rotations.

3.4.2 OOP Fuzzy Segmentation

In non-lit fuzzy classification, clipped geometry is rasterized by the fixed function rasterizing hardware which interpolates the texture coordinates for volume texture lookups. A fragment shader is then employed to lookup a voxel from the volume texture using hardware resident trilinear interpolation. This voxel is then used to address an additional texture map with a dependant texturing instruction. For post-classification a 1D lookup table is used (see Figure 3.21). Generally 256 entries is adequate to represent the detail within the volume as linear interpolation can be performed in hardware to return fractional results.

```
pixel fragmentShader(fragment, volume, transfer)
    voxel = volume(fragment.tex0)
    pixel = transfer(voxel.a)
```

Figure 3.21: OOP post-classification fragment shader

For pre-integrated classification an additional set of texture coordinates are issued to address the volume texture twice and obtain two sample locations (see Figure 3.22). These voxels are used to address the 2D pre-integrated classification table. The pre-integrated lookup tables are generally generated from a 256 entry 1D post classification transfer function, the resulting pre-integrated lookup table contains 256^2 entries.

```
pixel fragmentShader(fragment, volume, transfer)
    voxel1 = volume(fragment.tex0)
    voxel2 = volume(fragment.tex1)
    pixel = transfer(voxel1.a, voxel2.a)
```

Figure 3.22: OOP pre-integrated classification fragment shader

The shaders contained in this section provide no mechanism to compute early ray termination when the accumulated opacity along the ray reaches a threshold. This is due to the complete pipeline being used which offers no ability to stop the processing of fragments. Back-to-front compositing is also used since the fixed function hardware is used for blending which further restricts this function. Empty space skipping is also not possible with this approach since a rasterized fragment is used for each sample. A stage before rasterization of proxy geometry would have to be implemented for this feature to control which samples are issued fragments. This is currently not possible on current graphics hardware.

The benefit of this approach is the simplistic fragment shaders which allow high throughput in terms of the number of instructions required to correctly sample the volume dataset. However since no acceleration techniques can be included, this method degrades poorly

```
pixel fragmentShader(fragment, volume, transfer, light, textureMatrix)
    voxel = volume(fragment.tex0)
    normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
    light.diffuse = transfer(volume.a)
    pixel = lighting(normal, fragment.tex2, light)
    pixel.a = light.diffuse.a
```

Figure 3.23: OOP lit post-classification fragment shader

```

pixel fragmentShader(fragment, volume, transfer, light, textureMatrix)
    voxel1 = volume(fragment.tex0)
    voxel2 = volume(fragment.tex1)
    normal = (lerp(voxel1.xyz, voxel2.xyz, 0.5) * 2.0 - 1.0) *
        textureMatrix.inverseTranspose
    light.diffuse = transfer(volume1.a, volume2.a)
    pixel = lighting(normal, fragment.tex2, light)
    pixel.a = light.diffuse

```

Figure 3.24: OOP lit pre-integrated fragment shader

with increasing viewport and volume dimensions. This method is also computable in 1 pass alone. Lit variations for post-classification and pre-integrated classification are detailed in figures 3.23 and 3.24. These shaders are provided as a comparison to the non lit variants to show how more instructions begin to complicate the simple fragment shaders and introduce heavy burden. Dynamic branching on more recent graphics processors can be used to skip expensive functions such as lighting, however incur a cost when taking each branch.

3.4.3 OOP Binary Segmentation

There are two methods of opaque lit iso-surface rendering, binary segmentation in the fragment shader using conditionals (see Figure 3.25) and using the alpha test to perform the segmentation step (see Figure 3.26). The fragment shader method of segmentation compares incoming voxel scalar values to a predefined iso-value and performs lighting on these segmented voxels.

```

pixel fragmentShader(fragment, volume, isoValue, light, textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue.a)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif

```

Figure 3.25: OOP iso-surface fragment shader using conditionals

The alpha test performs lighting on all incoming samples and sets the fragment's alpha value to the voxel's scalar value. Fragments that do not contribute to the iso-surface are later discarded via the alpha test. Blending is not performed in hardware as the first hit of a lit iso-surface will occlude all other samples along the ray. In contrast semi-transparent iso-surface rendering requires blending to be enabled and the same fragment shader can be utilized providing the final alpha value can be controlled.

```

pixel fragmentShader(fragment, volume, light, textureMatrix)
    voxel = volume(fragment.tex0)
    normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
    pixel = lighting(normal, fragment.tex2, light)
    pixel.a = voxel.a

```

Figure 3.26: OOP iso-surface fragment shader for alpha test method

Interpolated iso-surface rendering is accomplished by segmenting incoming slab samples against a weighting table (see Figure 3.15) in the fragment shader. A generic interpolated iso-surface approach is achieved through weighting tables since single lit iso-surfaces can be extended to multiple iso-surfaces with no extra complexity. This allows for multiple iso-surfaces and provides weights for interpolating the gradients within a slab to calculate the exact iso-surface position. Multiple iso-surfaces without slab consideration reduces to lit post-classification.

```

pixel fragmentShader(fragment, volume, weight, colour, light, textureMatrix)
    voxelF = volume(fragment.tex0)
    voxelB = volume(fragment.tex1)
    wght = weight(voxelF.a, voxelB.a)
    if (wght > 0.0)
        normal = (lerp(voxelF.xyz, voxelB.a, wght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        light.diffuse = colour(voxelF.a, voxelB.a)
        pixel = lighting(normal, fragment.tex2, light)
        pixel.a = light.diffuse.a
    else
        discard
    endif

```

Figure 3.27: OOP multiple interpolated iso-surface fragment shader

Generally OOP algorithms under-sample the iso-surface when considering one sample location and produce stair-casing artifacts. This method produces more accurate renderings than single sample methods and allows the number of samples to accurately reproduce an iso-surface to be greatly reduced. The algorithm is more expensive to compute since four texture fetches are necessary, however the gain in quality against the number of sample locations allows a relative increase in speed to accurately reproduce an iso-surface.

This method can not include any accelerations other than blending being disabled through the graphics pipeline and additionally using dynamic branching on recent hardware to avoid costly functions such as lighting and matrix multiplication. Early ray termination is not performed as in the back-to-front rendering strategy, all iso-surface samples are lit and then copied into the frame buffer. Iso-surface samples further along the ray then replace these values and the last iso-surface sample encountered is used in the final image. Deferred shading is possible by rendering the gradient normals into the frame buffer rather than lit samples, however the shading of these iso-surface points with gradient normals must be done in a second pass. This is not taken into account since the description is limited to one pass. Empty space leaping is also not possible in this approach since each rasterized fragment is used as a sampling location only and does not contain information for multiple samples. An empty space leaping strategy would require that the rasterized fragments be generated with respect to a space leaping function which is not currently possible on graphics hardware.

3.4.4 OOP Results

The performance results for OOP rendering strategies are listed in table 3.6, rates are taken with no volume rotation to allow maximum performance during voxel texture fetches, a

decrease in performance can be observed for rotated views since memory alignment and caching in GPU hardware are affected. Non power of two texture sizes also incur a slight performance penalty due to the mip-mapping hardware implementation on GPU hardware. The results in this table represent speeds for the respective iso-surfacing techniques for OOP techniques since the conditionals within the fragment shader are set to condition code mode which executes every instruction.

Dataset (size)	Viewport	Slices	Clipping				No clipping			
			Pst	Pre	Iso	Int	Pst	Pre	Iso	Int
<i>BuckyBall</i> (32^3) see Figures 3.28 and 3.29	512^2	128	100	90	31	25	32	19	9	7
		256	58	45	16	13	16	9	5	3
		512	29	22	8	6	8	4	2	1
		1024	15	12	4	3	4	2	1	< 1
	1024^2	128	31	25	10	8	8	5	2	2
		256	16	13	5	4	4	2	1	1
		512	8	6	2	2	2	1	< 1	< 1
		1024	4	3	1	1	1	< 1	< 1	< 1
<i>CTHead</i> ($256^2 \times 113$) see Figures 3.30 and 3.31	512^2	128	97	52	30	22	23	17	9	7
		256	48	29	15	12	14	8	5	3
		512	24	15	7	3	7	4	2	1
		1024	12	8	4	1	3	2	1	< 1
	1024^2	128	30	21	10	7	8	4	2	2
		256	15	11	5	4	4	2	1	1
		512	8	6	3	2	2	1	< 1	< 1
		1024	4	3	1	1	1	< 1	< 1	< 1

Table 3.6: OOP rendering frame rates in frames per second, *Pst* is post-classification, *Pre* is pre-integrated classification, *Iso* is single iso-surface rendering and *Int* is interpolated iso-surface rendering. All frame rates are rounded downward.

Section 3.1.7 shows the cost of including dynamic branching. Lighting routines take the most amount of time in the fragment shader as a matrix multiplication is required to transform the normal into eye-space and additionally the ambient, diffuse and specular contributions computed. Skipping this routine can accelerate the throughput and produce faster frame rates where iso-surfaces are more sparse. Each hardware implementation performs approximately the same in full branch and non branch mode and maximal performance strategies vary for differing volume datasets and iso-values. The difference in performance between the two settings is negligible for the iso-surfaces tested which occupy around 50% of the volume. Very sparse volumes benefit from branching since the total number of cycles across all proxy geometry is reduced when including the dynamic overheads.

An additional consideration is the fragment discard that can affect the performance drastically depending on how the hardware deals with the instructions. On certain GPU implementations (4th generation ATI chipsets) there is a 50% speedup. Some current implementations will continue processing on a discarded fragment thought the remainder of the pipeline offering no speed-up and other implementations completely discard the fragment from any further processing.



Figure 3.28: BuckyBall dataset post-classification (a) to (d) and pre-integrated classification (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j)

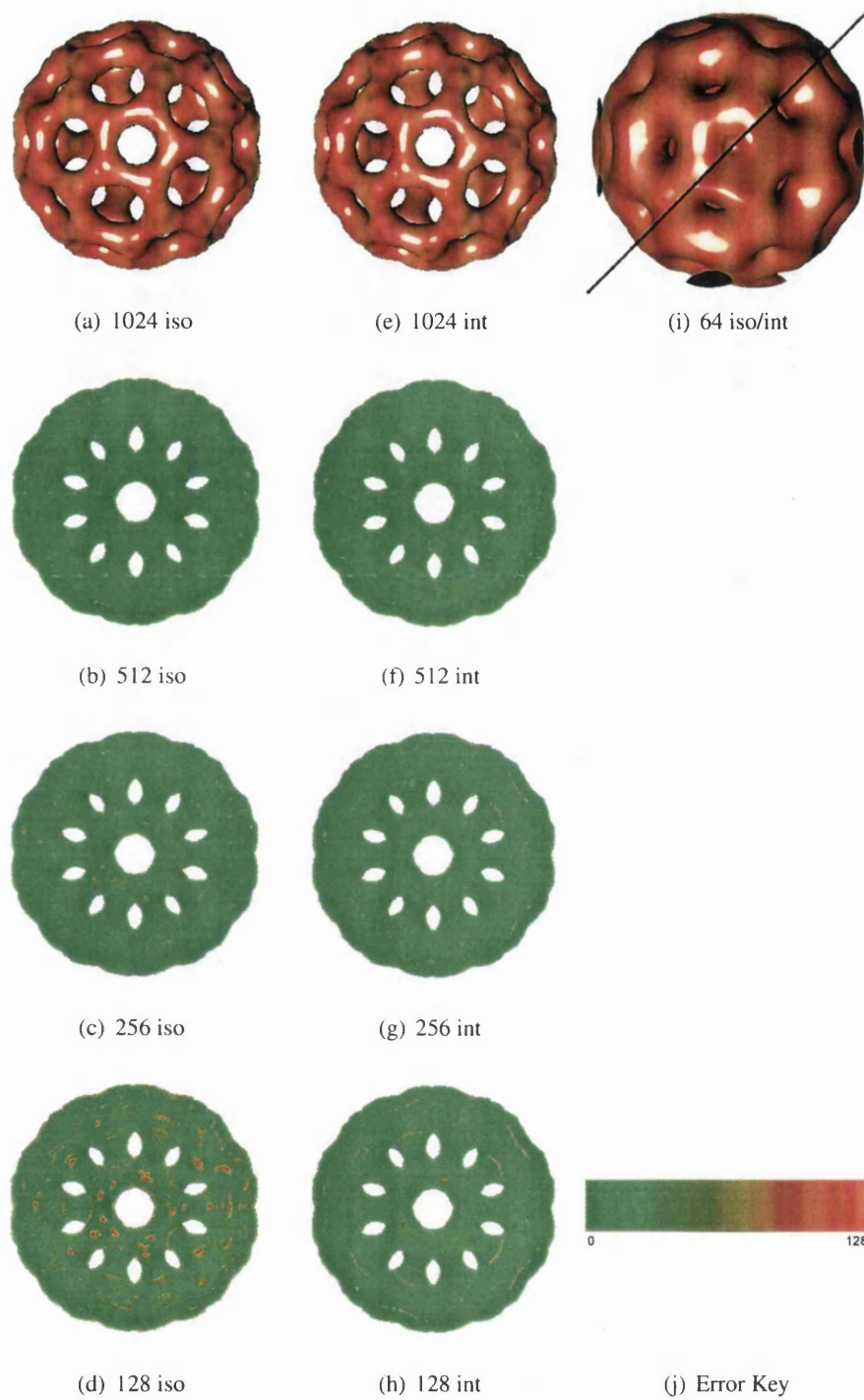


Figure 3.29: BuckyBall dataset iso-surface (a) to (d) and interpolated iso-surface (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

The purpose of the timings above are to compare the differing algorithms and are performed on the same GPU hardware unit. Setting the branch mode to condition code mode will yield a constant performance rate for all iso-surfaces. In general the alpha test mechanism of segmentation is analogous to setting condition codes on most hardware, although a slight performance gain might be noticed when using the alpha test as some burden is removed from the fragment shader and the pipeline is more balanced. Iso-surfacing with the alpha test can only describe one iso-surface at best and thus is demonstrated to compare performance. Both the alpha test segmentation and the fragment shader segmentation set to condition code mode perform lighting on all incoming fragments and represent the highest complexity for iso-surfacing. In contrast the fragment shader segmentation method set to use branch instructions has a lower complexity since lighting equations can be avoided, however branches can introduce more cycles into the overall rendering of all proxy geometry when the iso-surface chosen is not small.

The introduction of clipping planes observes an increase in speed based upon the execution of shaders on less fragments. Since clipping planes are easily added to the rendering method for any GPU hardware that is capable of volume rendering, this method represents the best overall approach to reduce fragments presented to the fragment processor. The slab rendering techniques incur a performance penalty since at least an additional texel must be fetched to obtain the front and back voxels. In pre-integrated rendering this is the only additional requirement of the algorithm since a classification lookup will require one texel lookup in both techniques.

Observing figures 3.28 and 3.30, the use of pre-integrated rendering techniques greatly increases the quality of the final output for equivalent sampling distances. In most cases including the outlined tests, pre-integrated outperforms post-classification for a trade off between performance and output quality. The transfer functions used for comparison contained small spikes to highlight post-classification schemes inability to accurately reproduce high frequencies.

Comparing Figures 3.28(d) and 3.28(h) it can be observed that pre-classification yields the best result, Figure 3.28(a) demonstrates the level of slicing a post-classification scheme requires to accurately represent the same detail. Observing Table 3.6 the performance throughput to represent the *BuckyBall* dataset with a high frequency transfer function correctly is much greater with pre-integrated techniques. Finally Figure 3.28(i) depicts a direct comparison between post-classification (top left) and pre-integrated classification (bottom right) for 64 sample slices.

For more complex volumes such as the *CTHead* dataset that contain higher frequencies between samples, it can be observed that pre-integrated classification outperforms post-classification to represent an accurate result (see Figure 3.30). The *CTHead* dataset is larger than the *BuckyBall* dataset and requires more sampling to due to its dimensionality and object complexity. Figure 3.30(a) represents the level of sampling that is required to obtain an accurate result with post-classification and Figure 3.30(g) represents the level of sampling required to depict a result to the viewer that contains all the necessary structure information without visible artefacts or under-sampled regions using pre-integrated techniques.

The performance for a representation that correctly portrays the objects detail without sam-

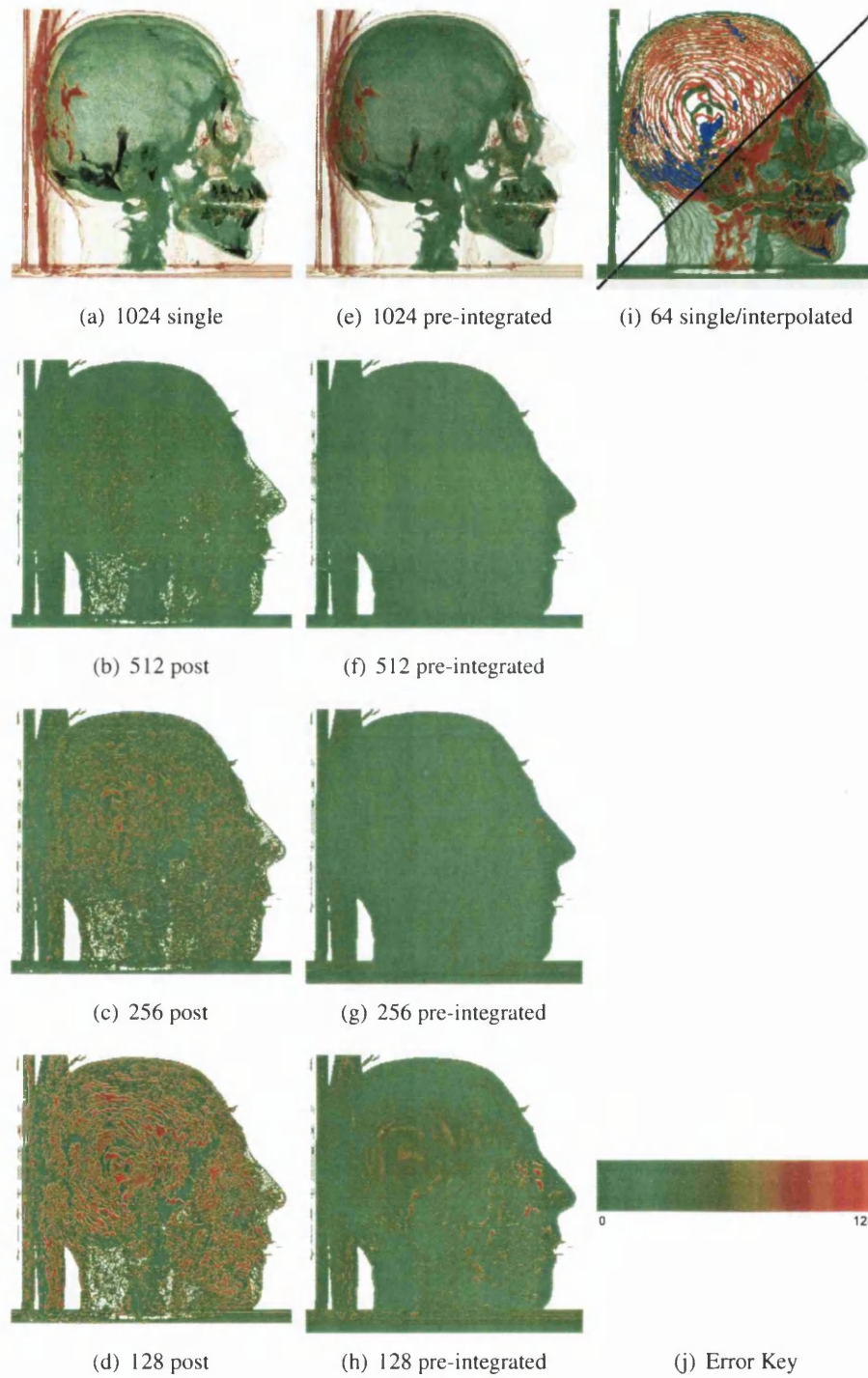


Figure 3.30: *CTHead* dataset post-classification (a) to (d) and pre-integrated classification (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

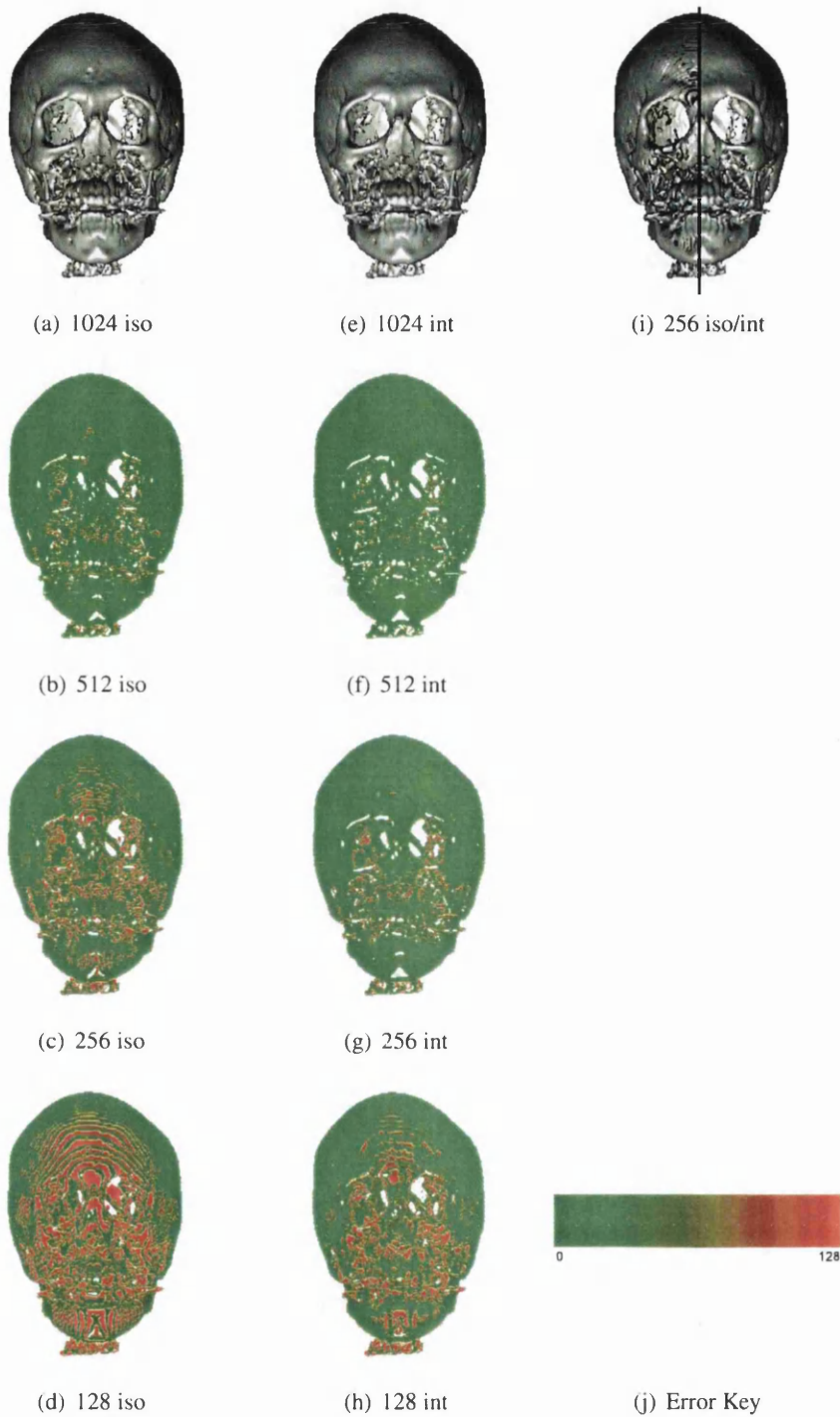


Figure 3.31: *CTHead* dataset iso-surface (a) to (d) and interpolated iso-surface (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

pling artefacts and under-sampling errors is again better with pre-integrated classification. Finally a more complex transfer function is depicted in figure 3.30(i) which compares post-classification (top left) and pre-integrated classification (bottom right) for 64 sample slices.

Figures 3.29 and 3.31 depict iso-surface rendering techniques for the *BuckyBall* and *CTHead* datasets respectively. Banding or stair-casing can be observed in Figures 3.29(d), 3.29(c) and 3.31(d). This is due to under-sampling the iso-surface in the z direction. The result of this under-sampling is usually a banding effect since fragments that are over the iso-value are rendered on an image aligned proxy slice when the previous slice's iso-surface does not run directly on the neighbouring fragment.

Interpolating with a weighting between front and back samples when considering a slab instead of a single sample can remove this effect by lighting exactly on the iso-surface and not using a gradient that is not computed in respect of the exact iso-surface. A comparison of accurate results 3.29(b) and 3.29(h) for single sampled locations and slab sampled locations shows that interpolating the gradient to lie on the iso-surface between a front and back sample yields greater performance. This effect is also noticed on higher frequency datasets such as the *CTHead* to a greater degree (see Figures 3.31(a) and 3.31(g)). Finally the two techniques are shown together in Figure 3.29(i) and 3.31(i).

3.4.5 IOM Rendering Framework

The multiple pass approach allows loops to be constructed to compute ray traversal of a volume dataset. Both front-to-back and back-to-front ordering can be achieved, however front-to-back ordering is considered here to take advantage of early ray termination, empty space leaping and adaptive sampling. The OOP rendering approach cannot incorporate these acceleration methods. This technique is a hybrid of the techniques described in section 3.4.1.

Ray setup for orthographic projection is computed with two initial passes. The bounding box of the volume is issued as a list of polygons which are passed to the GPU and rasterized into fragments for processing. This ensures unnecessary fragment processing outside the volume is avoided. The depth buffer is used for hidden surface removal so that there is only one fragment for each corresponding final image pixel. The subsequent passes compute the ray traversal through the volume. Each step along all rays are computed in parallel using the GPU fragment shading pipelines. An occlusion query is used to determine when all rays have been completely traversed. The occlusion query returns the number of fragments processed in each pass. There are three floating point buffers employed which are set to be render to texture targets. Reading and writing these texture maps cannot be performed asynchronously, and therefore each render target is set to read only or write only in a particular pass. Each render target's resolution matches that of the final frame buffer for display to ensure that texels in auxiliary buffers correspond to final image pixels in the frame buffer.

The first pass renders the bounding box geometry with the depth test set to leave the front faces (see Figure 3.32(a)). This pass is rendered into a floating point render target for use as a texture map in the next pass, and the values for each fragment represent the rasterized object space coordinates obtained from the corresponding vertex coordinates from the

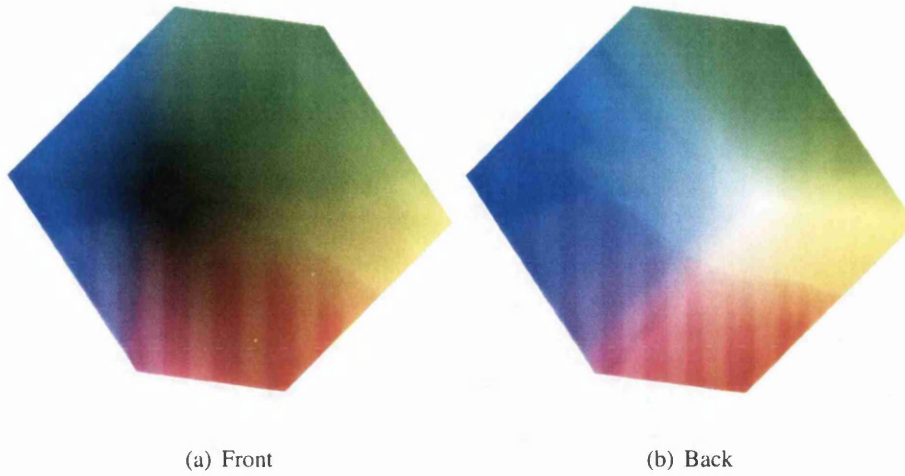


Figure 3.32: IOM proxy geometry, front and back faces with colours representing sample locations for ray entry and exit positions

bounding box geometry. Analogous to the explanation of OOP approaches, the viewport is set in the $[-1.75, 1.75]^2$ range. The vertex co-ordinates for the bounding box are issued in the $[-1, 1]^3$ range to allow for arbitrary rotations of the volume.

The second pass renders the bounding box geometry again with the depth test set to leave the back faces, which are the corresponding exit points of the volume (see Figure 3.32(a)). The texture map from the first pass is accessed to obtain the entry points into the volume. The ray direction and length is then computed in the fragment shader of the second pass and written into another texture map (see Eqn 3.10). The normalised ray direction is usually stored, however in this implementation the normalised vector representing the ray traversal through the volume bounding box is multiplied with the step size in order to reduce a computation at each sample location.

The parametric ray equation (see Eqn 3.10) is then directly computable in each sampling step by adding the step size adjusted direction vector, multiplied with the current sample index, with the starting location. Additionally if multiple samples are considered in each step, a simple addition to the current sample location can be performed. Subsequent passes that compute the ray traversal render the front faces of the bounding box to allow access to the starting point of each ray.

$$r(t) = s + dt \quad (3.10)$$

where t is the distance along the ray, s is the starting location and d is the direction vector.

$$dir(f, b, s) = (b - f)s$$

$$length(f, b) = |b - f|$$

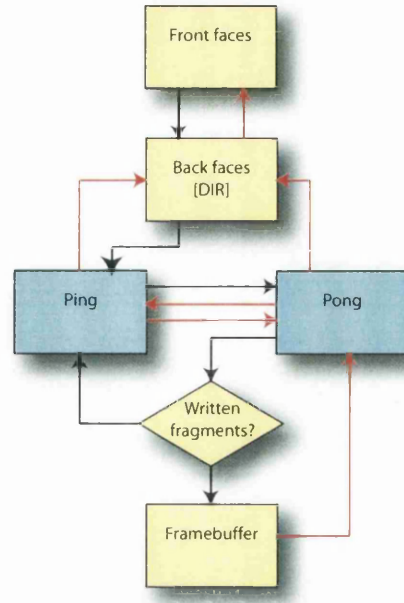


Figure 3.33: Ping-pong rendering scheme, red is texture access and black is flow path

```

fragment vertexShader(vertex, modelviewMatrix)
    fragment.tex0 = vertex.tex0
    fragment.tex1 = vertex.pos * modelviewMatrix.inversetrans

```

Figure 3.34: IOM vertex shader

where $f, b \in \mathbb{R}^3$ represent the co-ordinates of the entry and exit points of the volume bounding box (front and back), $s \in \mathbb{R}$ represents the stepping distance.

Two texture map render targets are used to compute blending via a ping-pong rendering scheme (see Figure 3.33). Both render targets are first initialized to the starting criterion of front-to-back compositing (see Eqn 2.19). At each pass a sample along the ray is computed and composited with the result of the previous pass or initial setting. This ping-pong rendering scheme can be used to compute direct volume rendering algorithms. Simply using the ping-pong rendering scheme alone allows increased accuracy during blending operations due to increased precision being used in computations. The fixed function hardware pipeline allows only 8 bit (16 bit by using offscreen buffers on 4th generation hardware) precision and the ping-pong scheme allows full 32 bit precision. However the overhead to use multiple render targets and switch between them is greater than an OOP method with off-screen buffers using native 16 bit blending operations. Intermediate passes can be interleaved between the main ping-pong passes to allow the inclusion of speed-up techniques.

It is possible to not use a ping-pong rendering scheme and blend directly into the frame buffer, however this reduces to 8 bit quantised blending per colour channel, and additionally incurs the extra burden of blending further along the pipeline which results in a decreased performance. Using the ping-pong rendering scheme to perform compositing, full 32 bit precision is available which is not possible by rendering into the frame buffer directly. The

previous description of OOP techniques (see section 3.4.1) mentions that it is possible to render into a floating point rendering target using two passes, however GPU hardware currently is only capable of blending up to 16 bit precision.

These increased precision blending operations in the specialised segment of the GPU pipeline incur an additional performance loss where the multiple pass ping-pong approach allows 32 bit precision by default as the fragment processors operate natively at 32 bits. In practice the rendering targets used in this algorithm have to be 16 bit floating point types as current GPU hardware does not allow 32 bit floating point rendering targets so the results are quantised, however this quantisation occurs after the result is computed and offers the best available precision on current GPU hardware implementations without computing the entire ray in a single fragment shader.

To allow for successful inclusion of the speed-up mechanisms of empty space leaping and early ray termination, intermediate passes are inserted after each main ping and pong pass. These passes are used to set the depth buffer and reject future fragments from main passes for processing based on the early z test. The early z test suppresses fragment processing for a given fragment by executing the depth test before fragment processing. Fragment shaders writing their own depth value cannot utilize the early z test because the result of the fragment shader might effect its passing or failing. Fragment shaders writing their own depth information must therefore be subjected to the depth test after fragment processing which is present in all GPU pipelines. Typically intermediate pass fragment shaders contain many less instructions and can be executed quickly. The more expensive shaders in the main passes are then successfully suppressed with a small overhead. This overhead can be negated because the complexity of ray sampling is reduced. The reduction in complexity is dependent on the volume dataset and additionally the transfer function.

In both original implementations [KW03, RGW⁺03] multiple ray steps are computed in one pass, this mechanism can greatly increase the throughput of the algorithm since altering the rendering target is a costly operation as the pipeline must finish before it occurs and additionally the geometry list reprocessed which results in a non constant usage of the GPU pipeline as no fragment shading will occur until the geometry is rasterized. In contrast the OOP techniques offer a constant throughput with minimal overhead at each stage of the pipeline, however more complicated algorithms (e.g. lighting) begin to swamp the fragment stage of the pipeline. A balance can be achieved by using differing quantities of multiple ray steps per pass. Older hardware generations only allow a maximum of four samples per pass because the maximum texture instructions in a single fragment shader are reached. More modern architectures allow many more texturing instructions which allows greater flexibility in balancing the pipeline. In practice around eight steps for each pass appear optimal. This is expected to increase with newer hardware implementations. A trade off is evident in this approach where the more steps taken in one pass can reduce the overheads of each pass but also extend the amount of samples processed before early ray termination can be performed.

```

pixel fragmentShader(fragment, volume, dir, res, transfer, step)
    ray = dir(fragment.wpos)
    blend = res(fragment.wpos)
    raypos = fragment.tex0 + (ray.xyz * step.aaa)
    if (length(ray.xyz * step.aaa) < ray.a)
        voxel = volume(raypos)
        output = transfer(volume.a)
        blend = composite(blend, output)
        raypos += ray.xyz
        .
        . // further samples
        .
    else
        pixel = blend
        pixel.a = 1.0
    endif

```

Figure 3.35: IOM post-classification fragment shader

3.4.6 IOM Fuzzy Segmentation

Fuzzy segmentation is achieved with the ping-pong rendering scheme enabled for blending. Standard post-classified 1D lookup tables (see Figure 3.35) and pre-integrated 2D lookup tables (see Figure 3.36) are both possible. The complexity of the pre-integrated rendering method is reduced since multiple ray steps are computed in one pass, which enables re-use of previously fetched voxel values. In general $n - 1$ less samples are required in a single pass that computes n steps along the ray.

Early ray termination is performed by examining the result of the previous main rendering pass. The texture map containing the result used in the last main pass is queried to fetch the texel corresponding to the fragment that was just processed to obtain the alpha component. Any update to this fragment will not have been composited into the next buffer, if the last pass filled the opacity, the depth buffer will still get updated to allow the next shader to pass. This will allow both buffers to contain the correct result, and each subsequent pass will result in no main fragment shader being executed. The texel is examined to determine if the opacity is full and no further compositing operations will effect the final result. Based upon this test the depth buffer is updated to control the next pass access to the resulting fragment via the early z test.

Additionally the direction texture map is queried to determine if the next sample will fall outside of the volume bounding box by querying the alpha component which contains the length of the ray. In the original implementation [KW03, RGW⁺03] only a single depth buffer is used with the depth test set to allow fragments with a greater depth. Since no depth output is changed in main passes, on passing the early z test, the depth buffer will be updated with the fragments own rasterized depth. This requires that the depth buffer is reset before each intermediate pass since the front face geometry being rendered with the intermediate shader pass will not pass the final depth test even though the fragment shader is executed. By setting the depth test function to greater or equal, the depth buffer does not require clearing, which increases performance.

Empty space leaping can be performed by adapting the intermediate passes to include an additional texture lookup into an empty space data structure. The original implementation

```

pixel fragmentShader(fragment, volume, dir, res, transfer, step)
    ray = dir(fragment.wpos)
    blend = res(fragment.wpos)
    raypos = fragment.tex0 + (ray.xyz * step.aaa)
    if (length(ray.xyz * step.aaa) < ray.a)
        voxel1 = volume(raypos)
        raypos += dir.xyz
        voxel2 = volume(raypos)
        output = transfer(voxel1.a, voxel2.a)
        blend = composite(blend, output)
        raypos += dir.xyz
        .
        .    // further samples
        .
    else
        pixel = blend
        pixel.a = 1.0
    endif

```

Figure 3.36: IOM pre-integrated classification fragment shader

[KW03] described using one level of a min-max octree structure at $\frac{1}{8}^{th}$ of the original resolution of the volume dataset. This texture map is queried for the minimum and maximum voxel values in the 8×8 neighbourhood. The minimum and maximum values are then used as dependent texture coordinates into an additional texture map that is generated against the current transfer function as a pre-processing step. This texture map is encoded to contain a binary value representing empty space or samples to be considered. When transfer updates are issued, this texture map must be recomputed. The intermediate pass is used to access this texture map from information obtained from the octree level structure. For this purpose the depth buffer must be cleared before intermediate passes since empty space skipping must be continually computed. This method requires that empty space leaping is examined at each pass. Consideration at each pass is required since no correspondence between ray direction and current octree level cell can be guaranteed without heavy overhead to compute exactly how much space can be skipped along the ray direction towards the boundary for the next octree level cell.

3.4.7 IOM Binary Segmentation

Fully opaque iso-surfaces do not require blending since the first sample encountered along a ray that contributes to an iso-surface will occlude all other samples along the ray. The multiple pass mechanism can accelerate opaque iso-surface rendering as a special case due to no blending being required and the inclusion of early ray termination. Additionally a localised version of deferred shading can be employed when analysing multiple samples in one pass, since only one shading calculation is required to correctly render the iso-surface. The looped passes mechanism can be cut down to allow one rendering buffer as the only requirement for the ping-pong scheme is to facilitate blending.

IOM iso-surface also offers an improvement in interpolated iso-surfacing since each main pass will compute multiple rays which allows reuse of previous values and reduces the amount of texture fetches performed.


```

pixel fragmentShader(fragment, volume, dir, step, isoValue, light,
    textureMatrix)
    ray = dir(fragment.wpos)
    raypos = fragment.tex0 + (ray.xyz * (step.aaa + samples))
    if (length(ray.xyz * step.aaa) < ray.a)
        voxel = volume(raypos)
        if (voxel.a < isoValue.a)
            temp = voxel;
        endif
        raypos -= dir.xyz
        .
        .    // further samples
        .
    if (temp.a > isoValue.a)
        normal = voxel.xyz * 2.0 - 1.0 * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif
else
    pixel.a = 1.0
endif

```

Figure 3.37: IOM iso-surface fragment shader

The multiple pass scheme outlined computes front-to-back ray traversal. Multiple sampling per pass in the fragment shader can be locally adjusted to compute back-to-front ordering to enable the first iso-surface hit to be recorded in a temporary variable (see Figures 3.37 and 3.38). This reverse sampling is necessary since no early ray termination is computed in the main pass, instead this is performed in the intermediate pass. Since no ping-pong rendering scheme is required an early ray termination can additionally stop the intermediate passes for a given fragment when the depth buffer is updated. These iso-surface rendering performance improvements are especially suited to older hardware that must evaluate all instructions and cannot issue dynamic branch instructions.

A localized deferred shading is computed in each main pass and therefore does not require that dynamic branching is computed in the main sampling of the ray because there are few instructions to consider. A dynamic branch can be computed for the lighting equation at the end of the shader on newer hardware which can offer an improvement in cases where the iso surface is located towards the back of the volume. Older hardware must use condition codes to skip this lighting equation which involves executing the instructions, however this still offers an improvement over the OOP strategy due to reduced numbers of lighting calculations along the ray. Full deferred shading can be performed by rendering the gradient normals into the frame buffer for an additional lighting pass. This will require clearing the z buffer and additional shaders.

Empty space leaping is performed by turning off the main passes by setting the z buffer accordingly with the intermediate pass shader. Any empty space leaping therefore must be computed in the intermediate shader. An octree structure can be used to perform empty space leaping analogous to the description of fuzzy classification (see section 3.4.6), however this structure must be queried at regular intervals since no correspondence from sampling position to the next octree cell along the ray can be guaranteed without costly intersection calculations. The intermediate shader must be kept as simple as possible to facilitate

```

pixel fragmentShader(fragment, volume, dir, weight, colour, step, isoValue,
    light, textureMatrix)
    ray = dir(fragment.wpos)
    raypos = fragment.tex0 + (ray.xyz * (step.aaa + samples))
    if (length(ray.xyz * step.aaa) < ray.a)
        voxelb = volume(raypos)
        raypos -= dir.xyz
        voxelb = volume(raypos)
        wght = weight(voxelf.a, voxelb.a)
        if (wght > 0.0)
            tempf = voxelf
            tempb = voxelb
            tempwght = wght
        endif
        .
        .    // further samples
        .
    if (tempwght > 0.0)
        normal = (lerp(tempf.xyz, tempb.xyz, tempwght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        light.diffuse = colour(voxelf.a, voxelb.a)
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif
else
    pixel.a = 1.0
endif

```

Figure 3.38: IOM *interpolated iso-surface fragment shader*

fast rendering.

Roettger *et al.* [RGW⁺03] use multiple render targets to encode the current sampling position, this mechanism allows for much greater flexibility in empty space skipping since an offset can be computed and rendered into this buffer to facilitate a complete skipping of this space. Distance fields may be rendered in this manner to take advantage of distance information contained in the sampled distance field (see section 3.2.4). However the original implementation does not compute empty space skipping. No wasted intermediate passes are necessary with this approach since the offset can be performed for each sample in the main shader. Updated sampling conditions can then be written into the ray buffer along with blending into the image buffer with multiple render targets.

3.4.8 IOM Results

The visual quality of direct volume rendering with the multiple pass strategy is similar to that of OOP approaches. The only difference being that when the volume is under-sampled, artifacts appear non-uniform since rays are not started from the same plane which results in samples not being taken on equidistant planes throughout the volume. Starting the rays from differing planes does produce more meaningful renderings when slightly under-sampling the volume since the perception of depth is not interrupted by apparent striping effects in the z direction. The quality of rendering is similar to the previous examples in Figures 3.28 - 3.31.



Dataset (size)	Viewport	Samp	Pst	Pre	Iso	Int
<i>BuckyBall</i> (32^3) see Figures 3.28 and 3.29	512 ²	128	38	35	74	55
		256	22	18	43	32
		512	12	10	24	17
		1024	4	5	12	9
	1024 ²	128	7	7	10	7
		256	4	3	5	4
		512	1	1	2	2
		1024	< 1	< 1	1	1
<i>CTHead</i> ($256^2 \times 113$) see Figures 3.30 and 3.31	512 ²	128	36	34	70	56
		256	20	18	40	30
		512	11	10	22	16
		1024	5	5	11	8
	1024 ²	128	8	7	9	7
		256	4	4	5	3
		512	2	2	2	2
		1024	1	1	1	1

Table 3.7: IOM frame rates in frames per second, *Pst* is post-classification, *Pre* is pre-integrated classification, *Iso* is single iso-surface rendering and *Int* is interpolated iso-surface rendering. All frame rates are rounded downward.

The performance results are listed in Table 3.7, rates are taken with no volume rotation to allow maximum performance during voxel texture fetches, a decrease in performance can be observed for rotated views since memory alignment and caching in GPU hardware are affected. This technique may also process more fragments. With the inclusion of early ray termination this value could increase as every different view of the volume can affect this property. Non power of two texture sizes also incur a slight performance penalty due to the mip-mapping hardware implementation on GPU hardware.

A further increase in speed could be included by negating to process the first two passes if the viewing parameters do not change per frame, however to compare this strategy to others the first two passes are computed for every frame. The results here are computed without empty space leaping since for direct volume rendering and iso-surfacing approaches, the octree is sampled at every intermediate pass which introduces a large texture lookup overhead and additionally distance field space leaping is not considered for iso-surfaces as multiple render targets are required which further adds to the burden of computation. These measures should be considered for very sparse datasets and large scale rendering problems, the general rendering problems explored in this thesis perform slower with the inclusion of these strategies.

Multiple passes introduce overheads to consider when processing each sample. These overheads are required to correctly perform blending and ray stepping, extra texture lookups are required to fetch the previous passes result for blending and the ray direction and step. Additional instructions are also required to compute blending and additionally to identify rays that have left the volume bounding box. Therefore the instructions to be executed by this algorithm is greater than the more simplistic OOP. In worst case scenarios such as when there

are no voxels to render and no early ray termination is performed, multiple pass algorithms are significantly slower than the OOP counterpart. The inclusion of empty space leaping can accelerate this case significantly, the worst case then becomes when every voxel must be considered and no early ray termination is possible. The rendering of the front faces of the volume bounding box to reduce fragments to process can be seen as an analogous to clipping planes for the volume bounding box in OOP techniques and offer no improvement. With the inclusion of empty space leaping however, multiple pass algorithms can perform better than the constantly performing OOP approaches.

Multiple pass approaches also benefit from not requiring a dynamic loop instruction as the loop is created through the multiple pass approach, conditional expressions are generally simplistic and do not contain many instructions and thus can be evaluated in condition code mode. In some cases, such as locally deferred shading strategies a dynamic branched conditional expression can offer an improvement in rendering sparse datasets.

3.4.9 IOS Rendering Framework

The IOS approach relies on newer hardware (5th generation, NVIDIA 6800) that is capable of dynamic branch instructions in the fragment shader. The entire algorithm is computed in a single pass within the fragment shader which allows greater flexibility and reduced overheads. No intermediate passes are necessary with auxiliary buffers which increases performance due to no switching between rendering buffers and reading in previous results using expensive texture instructions. The pre-integrated classification and multiple iso-surface algorithms also benefit from reduced texture lookups analogous to IOM rendering (see section 3.4.5). In addition the memory footprint of this approach is restricted to lookup tables and volume datasets since the inclusion of acceleration techniques can be computed locally without an increase in memory reserved for viewport sized buffers.

The computed precision is also the best possible since no quantising occurs for blending operations as the blending is computed locally in the fragment shader. This is analogous to IOM rendering, however since a maximum of 16 bits can be stored in texture maps currently, the algorithm quantises the blending operations for every n samples that is made in one pass. The OOP approach also quantises at every sample due to blending directly into the frame buffer (see section 3.4.1). The precision with this approach can be increased, however in practice makes the algorithm more complicated by using off-screen rendering targets as in the multiple pass approach. The greatest precision with this approach is currently 16 bits. In addition to these advantages over the other algorithms, IOS volume rendering offers early ray termination, empty space leaping and adaptive sampling without the overhead of computing intermediate passes to perform these functions, which in most real world cases does not provide a large speed-up because of the heavy buffer overheads and additional texture fetches involved.

The fragment shader loop instruction is only capable of providing 256 iterations at most since the loop counter is 8 bits. Stegmaier *et al.* [SSKE05] allow for more samples to be taken along the ray by nesting two loops to allow 65536 iterations. Break instructions are provided to exit the loop early in cases where the maximum number of iterations is not required, or early ray termination is to be performed. This method does rely on the

break instructions being honoured and only recent hardware and driver implementations (5th generation, NVIDIA 6800, Release 80 drivers) provide this mechanism. The overall performance of shaders that include loops depends heavily on the number of instructions to be executed inside the loop. Further branch instructions inside the loops also affect the performance of the overall algorithm. Since any branch currently has a high overhead (see section 3.1.7) it is necessary to balance the pipeline with further samples as an analogous to a speed-up mechanism in IOM techniques (see section 3.4.5). This approach reduces the number of cycles that loops require by computing more samples for every expensive branch instruction.

Considering the necessary pipeline balancing to achieve maximal throughput and the careful choice of condition code registers or branch instructions for bodies of conditionally executed instructions, it is clear that there is an optimal number of instructions to pad each iteration of a loop. In practice this is more than one sample in most volume rendering approaches such as fuzzy classification and iso-surface rendering. These extra instructions per loop are built up using more samples in a single loop and thus provide a neat solution to one of the problems with this method. That is the maximum number of samples possible with a single loop instruction. By including 4 samples in a loop the 8 bit loop counter that allows a maximum of 256 iterations will now compute 1024 samples along a ray. Increasing this to 8 samples per loop instructions allows 2048 samples along the ray, which in most real world applications is more than sufficient. The following descriptions of each volume rendering technique are computed with one loop instruction and several samples per iteration.

For comparison to the IOM techniques in section 3.4.5, the same mechanism for computing the ray direction is presented. This allows many less fragments to be considered as only the volume bounding box is rendered. This mechanism can be computed at the start of the fragment shader for both orthographic and perspective rendering by transforming either the texture co-ordinates of an image aligned quadrilateral or alternatively a point passed as a uniform parameter for perspective volume rendering. Both of these methods use a image aligned quadrilateral to provide the fragments to be considered. The overhead of the method outlined in section 3.4.5 to provide a texture map for ray direction and using bounding box geometry texture co-ordinates is used here to reduce the overall fragments to process and compare the actual rendering times by performing the same ray setup. Generally the image aligned quadrilateral will perform better with an appropriate fragment kill instruction where fragments outside the bounding box are encountered. This speed-up does rely on the hardware terminating the fragment and not writing a blank colour into frame buffer memory. This faster performance is due to having one less texture instruction per fragment and additionally not performing any buffer swapping during the first two passes.

3.4.10 IOS Fuzzy Segmentation

Post-classification is computed with a 1D transfer function and considered for one sample per ray step. Figure 3.39 shows the single pass post-classification fragment shader that computes both early-ray termination and detects the ray leaving the volume bounding box. Full 32 bit precision is present for blending operations. Pre-integrated classification is depicted in Figure 3.40 a 2D pre-integrated classification table is used to compute slab rendering.

```

pixel fragmentShader(fragment, volume, dir ,transfer)
    direction = dir(fragment.wpos)
    blend = (0.0, 0.0, 0.0, 0.0)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(raypos)
        output = transfer(voxel.a)
        blend = composite(output, blend)
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(raypos - fragment.tex0) || blend.a > 0.98)
            break
        endif
    endwhile
    pixel = blend

```

Figure 3.39: IOS post-classification fragment shader

Two samples for each location along the ray are used to address the 2D texture map. A speed-up is achieved for this method since the previous sample can be used as a sample in the next ray location. This reduces the amount of texture fetches necessary to compute this algorithm. This mechanism also includes early-ray termination and detects the ray leaving the volume bounding box.

These speed-ups are the most efficient possible in the IOS case since the whole ray is computed in the fragment shader which allows half of the texture fetches to be removed in the case of slab sampling methods. Additionally the early ray termination can be locally in the shader which removes the burden of requiring an intermediate pass.

Empty space skipping can be included in IOS rendering by issuing a further texture map defining an octree level analogous to the description of IOM rendering (see Section 3.4.9). As with the previous explanation, this method does not allow a significant speed-up since conditionals are used to control sampling. A texture fetch is required per sample since empty space cannot be leaped effectively due to no guarantee that skipping the cell size will not skip over important samples. This is due to sampling positions not being in the centre of a cell.

3.4.11 IOS Binary Segmentation

Opaque Iso-surface rendering is achieved without blending operations being performed in the fragment shader (see Figure 3.41) by stepping front-to-back and detecting the first iso-surface intersection. This iso-surface intersection is then shaded in a deferred manner at the end of the fragment shader. No shading is performed if the iso-surface is not intersected by using a conditional expression. This offers the best available deferred shading approach since the whole ray can be sampled before any shading which requires additional passes in both OOP and IOM rendering. Additionally extra information is available such as position and weighting function results. This removes the requirement to perform additional texture fetches or store the ray position in additional buffers, such as IOM approaches which would

```

pixel fragmentShader(fragment, volume, dir, transfer)
    direction = dir(fragment.wpos)
    blend = (0.0, 0.0, 0.0, 0.0)
    rayPos = fragment.tex0
    voxelF = volume(raypos)
    rayPos += direction
    while (true)
        voxelB = volume(raypos)
        output = transfer(voxelF.a, voxelB.a)
        blend = composite(output, blend)
        voxelF = voxelB
        rayPos += direction
        .
        .    // further samples
        .
        if (direction.a < length(raypos - fragment.tex0) || blend.a > 0.98)
            break
        endif
    endwhile
    pixel = blend

```

Figure 3.40: IOS pre-integrated classification fragment shader

require more multiple rendering target buffers to be stored and rendered to account for position.

Early ray termination and termination upon leaving the bounding box is also computed. Opaque slab style iso-surface rendering benefits from less texture lookups due to the previous sample being available without an additional texture fetch (see Figure 3.42). This also applies to the multiple opaque iso-surface rendering technique. Additional instructions can be included to perform blending between each sample with conditional lighting to compute semi-transparent volume rendering. In practice this is more expensive since dynamic conditionals are required to control whether lighting is applied to samples encountered.

Empty space skipping can be performed with IOS fragment shading with relative ease. The position along the ray can be adjusted by altering the ray step increment. This does not intrinsically suit an object order data structure such as an octree, however proves useful for distance field rendering with empty space leaping (see section 3.2.4). The equivalent empty space skipping in IOM approaches requires additional multiple rendering buffers to update the sample locations whilst performing empty space skipping for distance fields. Empty space leaping in slab based rendering approaches does require additional overhead since the space between respective samples is unknown. Therefore the reuse of previously fetched results cannot be employed and it is possible that extra texture fetches are necessary. However the scalability of performing empty space leaping on larger rendering problems is improved.

3.4.12 IOS Rendering Results

The visual quality of the techniques presented for IOS techniques offer the best available on current GPU hardware due to full 32 bit precision being used when using tri-linear interpolation and pre-integrated classification and post-classification. Higher order techniques not covered here such as tri-cubic interpolation will also exhibit better visual results due to full

```

pixel fragmentShader(fragment, volume, dir, isoValue, light, textureMatrix)
direction = dir(fragment.wpos)
rayPos = fragment.tex0
while (true)
    voxel = volume(rayPos)
    if (voxel.a > isoValue.a)
        break
    endif
    rayPos += direction
    .
    .    // further samples
    .
    if (direction.a < length(rayPos - fragment.tex0)) {
        break
    endif
endwhile
if (voxel.a > isoValue.a)
    normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
    pixel = lighting(normal, fragment.tex1, light)
endif

```

Figure 3.41: IOS *isosurface fragment shader*

```

pixel fragmentShader(fragment, volume, dir, weight, colour, light,
    textureMatrix)
direction = dir(fragment.wpos)
rayPos = fragment.tex0
voxelf = volume(raypos)
rayPos += direction
while (true)
    voxelb = volume(rayPos)
    wght = weight(voxelf.a, voxelb.a).a
    if (wght > 0.0)
        break
    endif
    voxelf = voxelb
    rayPos += direction
    .
    .    // further samples
    .
    if (direction.a < length(rayPos - fragment.tex0))
        break
    endif
endwhile
if (wght > 0.0)
    normal = (lerp(voxelf.xyz, voxelb.xyz, wght) * 2.0 - 1.0) *
        textureMatrix.inverseTranspose
    light.diffuse = colour(voxelf.a, voxelb.a)
    pixel = lighting(normal, fragment.tex1, light)
endif

```

Figure 3.42: IOS *interpolated isosurface fragment shader*

32 bit blending. Generally the volume datasets used are 8 or 16 bit precision which means the effects are not visually noticeable on most small viewport renderings. The quantised versions do introduce small artefacts that this technique avoids. The images presented in Figures 3.28 - 3.31 can be considered in most circumstances comparable to those of this technique.

Dataset (size)	Viewport	Samp	Pst	Pre	Iso	Int
<i>BuckyBall</i> (32 ³) see Figures 3.28 and 3.29	512 ²	128	57	51	60	30
		256	24	24	33	17
		512	16	15	18	10
		1024	7	7	10	5
	1024 ²	128	7	8	19	5
		256	4	5	10	3
		512	2	2	5	2
		1024	1	1	2	1
<i>CThead</i> (256 ² × 113) see Figures 3.30 and 3.31	512 ²	128	54	47	55	27
		256	22	22	22	16
		512	15	14	17	9
		1024	7	8	9	5
	1024 ²	128	8	7	10	6
		256	4	4	5	3
		512	2	2	3	1
		1024	1	1	1	< 1

Table 3.8: IOS frame rates in frames per second, *Pst* is post-classification, *Pre* is pre-integrated classification, *Iso* is iso-surface rendering and *Int* is interpolated iso-surface rendering. All rates are rounded down.

The timings for this technique are depicted in Table 3.8. Frame rates are taken with no volume rotation to allow maximum performance during voxel texture fetches, a 10% to 15% decrease in performance can be observed for rotated views since memory alignment and caching in GPU hardware are affected. With the inclusion of early ray termination this value could increase as every different view of the volume can affect this property. Non power of two texture sizes also incur upto a 10% performance penalty due to the mip-mapping hardware implementation on GPU hardware being optimised for power of two sizes for caching.

A 50ms increase in speed per frame could be included by negating to process the first two passes if the viewing parameters do not change per frame for a 512² viewport since the first two passes run at over 100 frames a second when isolated, however to compare this strategy to others the first two passes are computed for every frame. Additionally no empty space leaping is performed for these results as an octree based data structure requires constant sampling and introduces overheads by adding additional cycles on the GPU for each loop (see 3.4) since dynamic branching must be employed instead of condition code branching within the ray sampling loop. Distance field space leaping is explored later in this thesis.

The main expense with this approach is the loading of the fragment shading stage of the hardware pipeline with lengthy fragment shader code and in addition the necessity to in-

clude dynamic branching in certain cases such as looping and conditional expressions that exhibit large blocks of instructions. However the scalability is improved over other techniques and future hardware will improve the costs involved with dynamic conditionals.

3.5 Summary

In this chapter a variety of volume rendering techniques for GPU's are explored for both fuzzy and binary segmentation tasks. This chapter reviews existing strategies for the volume rendering pipeline on the GPU, compares these methods with analysis of performance and quality with implemented hybrids of these algorithms. Accurate measurements of performance have been taken for hybrid implementations of well known rendering approaches with a description of hardware specific issues and limitations for older and state of the art hardware. Improvements in several areas have been outlined for inclusion in implementations of GPU volume rendering pipelines based on these techniques. Further investigation is given to improvements and performance increases for each volume rendering technique. The implemented algorithms compute standard volume rendering problems and act as a basis for further expansion and additional rendering techniques.

The quality of rendering is greatly increased by using slab based rendering strategies, and as a result less samples need to be considered for a quality image which gives rise to the increased performance. In general for each separate classification task, slab rendering has proven to be more effective in terms of both quality and speed. The additional overheads are pre-processing problems that occur on parameter changes such as iso-value or transfer function changes. Interactive rates are still observed in these cases.

The importance of balancing the pipeline for maximal throughput was outlined with techniques to achieve this effectively. The difference between condition codes and branch instructions is discussed as a balancing mechanism along with multiple samples per pass / iteration. In general, a formula to arrive at a specific formation of these techniques for a specific hardware implementation is difficult as every hardware implementation and hardware generation will handle these important instructions differently. It is highly probable that future generations will include more sophisticated pipelining strategies to counteract branch instruction overheads. Equivalent processors for CPU's now take advantage of out of order execution and sophisticated branch predictions, currently the GPU processors are far more simplistic in nature, however future implementations will strive to combat such issues as programmers wish to implement more advanced fragment shaders. The volume rendering problem on GPU's continues to be at the front of shader technology, as each new generation allows more to be performed effectively. However the outlined techniques will continue to be the basis of hardware accelerated volume rendering tasks.

The OOP algorithm is the most straightforward to implement. This method also works on all hardware generations capable of 3D texture mapping with trilinear interpolation. Older hardware generations can also be used by adapting the 3D texturing techniques for 2D slice texturing techniques. These 2D algorithms are faster than their 3D counterparts due to the memory alignment of the 2D texture maps as in the shear-warp rendering algorithm. The OOP technique offers the best multi-platform algorithm and additionally in some cases the

fastest due to minimal instructions. The respective quality is limited with this approach however since the blending hardware is used, and without rendering into an off-screen buffer, is quantised to 8 bits per sample. Multiple passes have been discussed to increase the quality of blending by not directly blending into the frame buffer, however these techniques alter the performance of this approach which is used in this thesis as a bench mark for encoding the minimum instructions to compute a problem. This approach does not scale well over the volume rendering problem as no speed-up mechanisms can be introduced.

The IOM technique has been shown to be most effective whilst iso-surfacing to take advantage of locally deferred shading and additionally early-ray termination. Most real world (such as low opacities contained in a transfer function to view general volume construction) fuzzy classification tasks do not allow early ray termination as full opacity is not reached by the end of the ray. This technique is the most complicated to implement and relies heavily on specific hardware features that are only present with recent generations. The overheads due to multiple buffers, multiple rendering targets and multiple passes often outweighs the algorithm's reduced overall complexity in the average cases. The quality of the output is marginally better than the OOP approach due to improved quantisation during blending. This algorithm does scale better than its OOP counterpart over the volume rendering problem, however requires many additional attributes to achieve this performance increase.

Finally the IOS algorithm offers a straightforward algorithm to compute image-order techniques compared to its multiple pass counterpart. This method does require a new generation (5th generation, NVIDIA 6800) of GPU hardware, however is greatly simplified for the same problem. The quality of the output is the best available on current state of the art GPU's (5th generation, NVIDIA 6800) and the performance is better than multiple pass techniques due to the inherent simplification in terms of multiple passes and additional texture maps in several cases. This method scales well over the volume rendering problem since it can benefit from speed-up techniques. Additionally the memory footprint of this approach is improved over the IOM counterpart and is considered more efficient in this regard.

OOP techniques are best for cross vendor and cross generation implementation of volume rendering applications and is considered a standard rendering platform for the GPU. Thus any simple volume rendering method available on the GPU should be implementable using this approach. The speed of this approach is due to its simplicity and simple fragment shaders. The IOM strategy outperforms OOP methods in iso-surfacing alone (*CTHead* 256 samples, IOM 40fps, OOP 22fps), and in general is superseded by the single pass approach. The single pass approach will improve with each future generation as hardware processors become smarter and compilers better. This is due to the branching computation on current hardware and the obvious overhead of multiple passes and multiple output buffers for the multiple pass counterpart. Therefore the IOS volume rendering technique should be considered for image-order techniques as over time it will enable a more scalable approach, it's only requirement being that better dynamic branching hardware is introduced.

Chapter 4

Volume Texture and Hypertexture

Contents

4.1 GPU Procedural Texture Primitives	112
4.2 Solid Texturing Volumetric Objects	120
4.3 Volume Hypertexture	131
4.4 Animation Techniques	144
4.5 Summary	151

This chapter explores real-time volume graphics techniques to add rich detail to complex volumetric objects. The approaches outlined in this chapter utilise standard volume datasets and distance field volume datasets which allow many other graphical representations to be imported through voxelisation. This chapter introduces GPU volume rendering algorithms to compute solid texturing and hypertexture effects using procedurally generated texture descriptions on the fly.

Surface based techniques such as texture mapping [Cat74, Cat75, Bli78a] are commonly used to enhance the appearance of an object's surface by adding complex image information. Methods such as bump mapping [Bli78b], displacement mapping [Coo84] and environment mapping [BN76] have been developed to add detail to surfaces without a serious impact on rendering speed. These techniques however are incapable of truly representing natural occurring properties due to many real world substrates having no surface definition.

Volume graphics allows the modelling and rendering of semi-transparency, naturally occurring and amorphous phenomenon as well as solid objects. These attributes over surface graphics are exploited to enable complex object definition and texture to be applied to volumetric primitives.

Procedural texturing describes the generation of realistic natural patterns with algorithms. No texture artist or photography is used to define a texture map, instead functions are evaluated and combined to synthesise patterns that model real world substrates. Generally some form of stochastic function is used to introduce frequencies that represent randomness encountered in nature. Procedural texture synthesis has been shown to be an important fundamental graphics technique for describing many object properties [Per85]. Interactive

procedural texture synthesis is explored for describing surface texture with *solid texturing* which alters a surfaces colour definition, and higher-order object definition with *hypertexture* which extends the definition of an object to include a malleable soft-region outside of it is defined surface. This malleable soft-region removes previous restrictions in texturing a solid surface. Often natural surfaces such as hair, fire and smoke cannot be represented with simple surfaces or texturing of surfaces as they are too complex.

These new interactive techniques can be employed in many fields where internal and external object detail is required. One such application is the games industry ¹, where effects such as fire, smoke, clouds and melting are extremely common and difficult to represent intuitively. Having internal object data with such effects also provides better playability when deformations occur in game play. There are numerous other applications in computer graphics where modelling and rendering of semi-transparent and naturally occurring properties are required to describe the desired object.

Providing procedural texturing primitives to the GPU are described in section 4.1. Solid texturing an object's surface is described in section 4.2 and hypertexture is explored in section 4.3. The animation of procedurally textured objects is explored in section 4.4 and finally section 4.5 provides a summary.

4.1 GPU Procedural Texture Primitives

Noise is a common method of introducing randomness into a procedural texture and is the chosen fundamental procedural texturing primitive for inclusion in a procedural texturing framework. Many procedural textures can be constructed without the inclusion of stochastic properties, however are usually too uniform to correctly approximate real world aesthetics and structure. A noise function should conform to the following properties:

- Repeatable pseudo-random function of input
- Known range of output (usually $[-1, 1]$)
- Statistical invariance under rotation, translation and scaling
- A narrow bandpass limit in frequency
- Non-repeating pattern towards infinity

These properties ensure that when modelling behaviour changes, no visible artifacts are present in the rendering and the underlying pattern is not affected in any manner. A noise function that satisfies these properties is acceptable for general procedural modelling use and can provide extremely aesthetically pleasing results. These properties allow it to be used in general modelling situations and allow any manipulation to be uniform throughout the domain. Perlin noise is chosen since its properties obey these conditions and additionally the output is aesthetically consistent with smoothed natural white noise. Perlin noise is also a controllable and efficient function to provide stochastic behaviour in comparison to other techniques outlined.

¹GDC05 covered the emergence of simple volumetric effects

Perlin noise can be computed from $1, \dots, n$ dimensions and returns a scalar in the $[-1, 1]$ range.

$$\text{noise}(p) = \text{pseudo-random scalar} \quad (4.1)$$

where p is a coordinate in n space and the returned pseudo-random scalar is representative of the band-limited white noise being approximated.

Additional higher order procedural texturing primitives can be defined by combining noise functions. Turbulence is an example of a higher-order noise function which sums several contributions of noise in a sinusoidal manner to produce a high frequency noise function that appears to produce turbulent flows. The number of differing noise frequency contributions is referred to as octaves. The turbulence function does not directly consider flow directions. Turbulence also obeys the noise function properties as it is compiled from noise function primitives and therefore can be used as a general procedural modelling function.

$$\text{turbulence}(p) = \sum_i \text{abs} \left(\frac{\text{noise}(2^i p)}{2^i} \right) \quad (4.2)$$

where p is a coordinate in n space and i is the number of octaves of noise to sum.

There are two approaches to implementing the Perlin noise algorithm on the GPU, computed with instructions on the GPU such as a non rasterized *f-rep* or firstly pre-computing a rasterized approximation of the noise function for a limited domain. Computing the original function at run-time is denoted local function evaluation and pre-computation is denoted pre-computed evaluation. Since noise needs to be evaluated for samples during volume rendering, the noise function must be present during fragment shading. There is currently no hardware support for a noise algorithm on commercially available GPU's.

The two methods of texture based and procedural based noise are evaluated for performance requirements and aspects of a good noise implementation. The goal of this research into noise implementations on GPU hardware is to provide a good general noise implementation to the GPU for a procedural texturing framework. This noise implementation should satisfy any performance and visual quality trade-off to be utilised as a general primitive available to procedural texture artists. 3D noise is considered here as a platform although 4D noise should also be available for performing animation effects over time.

4.1.1 Pre-computed Evaluation

In pre-computed implementations, a noise block with a finite domain is constructed on the CPU and uploaded to the GPU as a texture map. Texture maps can be saved as volume datasets to minimise the amount of processing required at run-time or alternatively pre-computed at each invocation of an application to minimise storage requirements. The size of the texture map is important since maximal space must be left for other texturing or lookup table requirements in GPU memory and the texture should contain a detailed snapshot of noise. An acceptable size is around 64^3 for a 3D block of noise as this will contain a good approximation of an original noise implementation for the desired application for

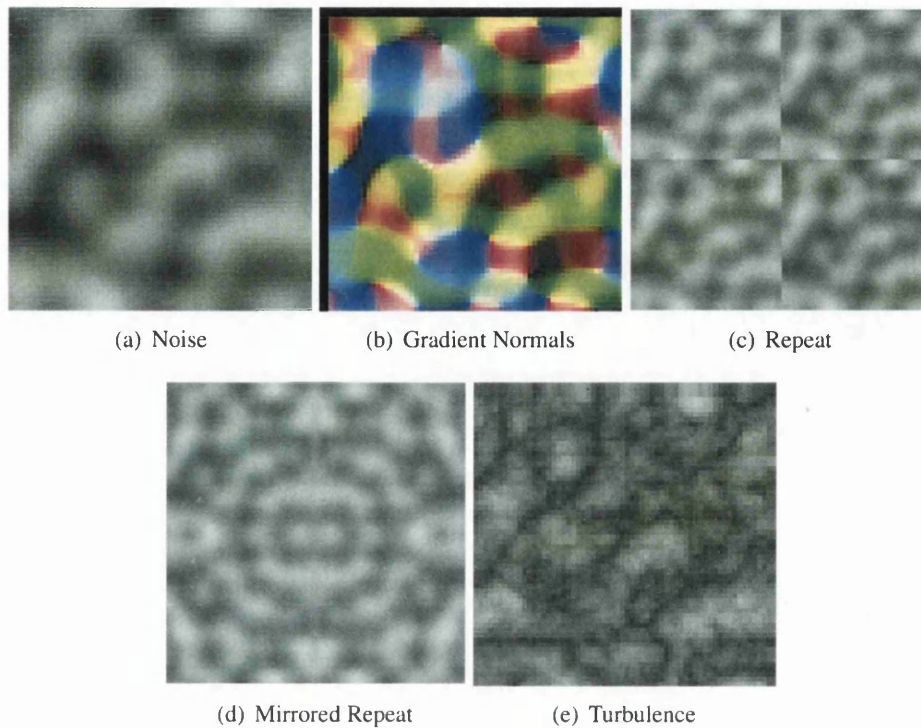


Figure 4.1: Examples of Perlin noise

a small to medium viewport. The texture can be accessed with trilinear interpolation to ensure a smoothed result, however if the viewport becomes too large, the stochastic appearance becomes less evident in the final image. This is due to the cubic interpolation described in the noise function being approximated from sample points already computed at non uniform grid locations. A noise volume is not restricted to the same dimensions as the volume dataset due to the hardware resident trilinear filtering available. Larger viewports and volume datasets benefit from larger texture based noise lookup tables, however there is an image quality and texture memory trade off to consider.

The fundamental problem with texture based noise lookup tables is when operating outside the defined texture co-ordinate range or texture domain. This is a common feature of procedural techniques that build up specific stochastic behaviour with multiple noise function components. Good noise implementations should appear random through an infinite domain without artifacts (see Figure 4.1(a)). Addressing a noise texture block can result in the noise visibly repeating instead of remaining random. The properties of the underlying noise implementation are lost during pre-processing into a texture block since addressing outside the texture block space will involve a clamp or repeat texturing operation. Texturing modes that support repeating textures in the hardware pipeline simply repeat the original texture map, which leads to a visible artifact at each border (see Figure 4.1(c)). One method of circumventing this problem is to use a mirrored repeat texture mode. This minimises the visual artifacts at the borders since a visible repeating pattern is observed, however the mirrored nature of the resulting noise pattern removes the stochastic randomized properties

(see Figure 4.1(d)).

This also becomes a problem when using noise gradients, visible repetitions and border artifacts are introduced (see Figure 4.1(b)). Generally the edges of the noise volumes bounding box has no gradients defined since gradient computation methods leave at least a one texel border (see section 2.4). These values can be approximated, however there is still an artifact issue with repeating or mirrored repeating since the one texel border that is approximated will look uniform when repetition mechanisms force two border definitions together.

Pre-computing the noise or higher order noise functions over a discrete grid in software produces a texture encoded with the final noise configuration (see Figures 4.1(a) and 4.1(e) respectively). Only one texture fetch is needed to utilize this pre-computed noise block providing the best performance possible. Therefore texture co-ordinates will not fall outside of range, removing bordering issues. Additionally greater image quality will be generated by using the precision available in software for similar volume sizes and viewports, despite this approximation being subject to trilinear interpolation of sub grid positions. The trade-off with this approach is the loss of detailed noise with increasing resolution and having to ensure the textures domain is adequate to describe each required noise pattern.

Constructing higher order noise primitives such as turbulence (see Figure 4.1(e)) using textures is possible via multiple texture fetches into a standard texture noise texture. This method suffers from bordering problems when addressing outside the texture co-ordinate range, and additionally suffers from not being truly random over an infinite domain. Since the precision on the GPU is lower in most cases than software, the image quality is reduced and noise appears less stochastic. Rendering speed is slower because additional texture fetches are required and additionally the GPU is responsible for any additional computation between texture fetches (such as the *abs* function when computing turbulence). When using a large enough noise domain to correctly encode higher order noise primitives, bordering issues are removed and a more random visual effect is created by computing the final function in the fragment shader. Resolution expansion restrictions are removed in this manner.

To solve the overhead of several texture fetches and bordering issues when constructing higher order noise primitives, differing frequencies of noise can be encoded in each separate colour channel of the texture map. This eliminates three additional texture fetches by utilizing the spare channels when constructing turbulence with four octaves. The image quality is similar to multiple lookups, however bordering artifacts are removed due to separate blocks being computed outside the texture address domain. This results in a larger amount of resident noise primitives since each channel of the texture contains noise at different frequencies. This removes the bordering artefacts when constructing higher order noise primitives and additionally allows differing noise frequencies to be present which drastically reduces the burden of ensuring a complete texture domain is provided.

The performance and visual results of these pre-computed noise and higher order noise techniques are discussed in section 4.1.3 where a comparison to a non-approximated locally evaluated noise function is presented.

4.1.2 Local Evaluation

The noise algorithm contains two lookup tables that are used to describe pseudo-random behaviour. These tables guarantee that the noise function will return the same result for the same input conditions which is an important property. The two tables are a permutation table that defines a list of random integers and the gradient table which is a list of random normalized gradient vectors. The permutation table in the original implementation contains 256 entries, with the entries being randomized integers in the $0, \dots, 255$ range. These permuted indices are used to lookup a randomized gradient in the gradient table. The gradient table also contains 256 entries with each entry being a unit length vector ($[0, 1]^3$). These gradients are chosen such that when assigned to grid points, the gradients at each corner of a grid cell will be different.

Both these tables cannot be efficiently computed in the fragment shader since a fixed (pseudo-random) permutation table cannot be computed in one fragment shader for a given fragment and kept in global variable memory for execution of other fragments. Additionally computing the gradient table involves a form of sphere mapping to describe unit length vectors emanating from the origin of a sphere. These mechanisms can be carried out with multiple passes through the graphics hardware and stored as lookup texture maps, however it is more efficient to pre-compute these values, store them as constants in source code.

An issue with implementation of Perlin noise on the GPU is therefore how to provide these permutation and gradient tables. The GLSL specification [KBR] outlines using arrays from uniform variable locations, however current graphics hardware for fragment shaders does not have the ability to do dynamic array addressing at run time, each address for lookup must be known at compile time. Vertex shaders do have this ability on current GPU hardware although all volume computations require per fragment processing and thus this method currently cannot be implemented.

Therefore to provide the permutation and gradient tables to the fragment shader, 1D texture maps are required. This method is not as efficient as providing an array of constants since it involves texture lookups into texture memory rather than local registers. Future GPU's should allow dynamic array access in local registers which will accelerate this algorithm.

Since the original function can be computed entirely in the fragment shader due to uploading the permutation and gradient tables, there are no restrictions on available noise frequencies and an infinite domain is provided. This approach however is expensive since standard noise in 3D requires around 10 texture fetches for noise. Computing 8 octaves of turbulence would thus require 80 texture fetches per fragment which is too expensive in practice.

Green [Gre05] provided a reference implementation of Perlin noise for GPU hardware. This implementation takes advantage of vector operations available in fragment processors to reduce the number of operations required by the algorithm. Additionally the permutation mechanism is precomputed into a 2D texture map to avoid fetching 2 permutation table entries. This reference implementation is used here as a comparison to pre-computed noise blocks in favour of Hart's [Har01] implementation of noise in pixel shaders using multiple passes. The reference implementation described is directly comparable to Perlin's [Per85] original noise implementation.

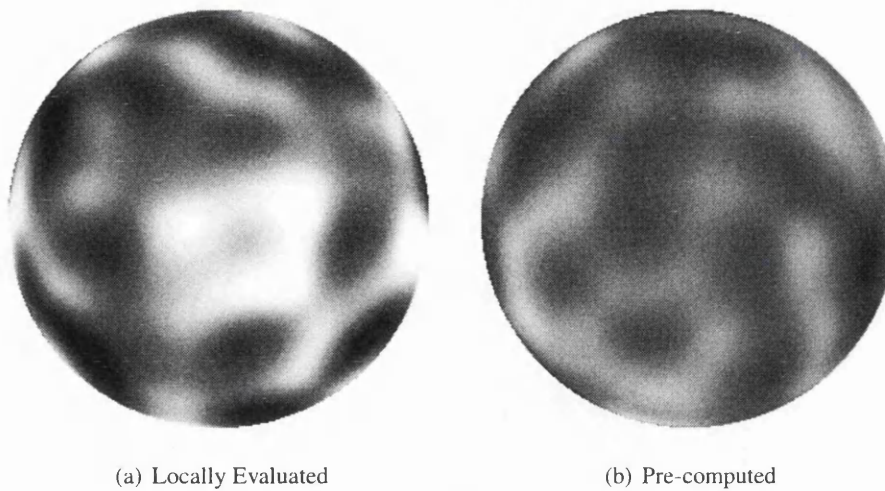


Figure 4.2: Examples of noise implementations

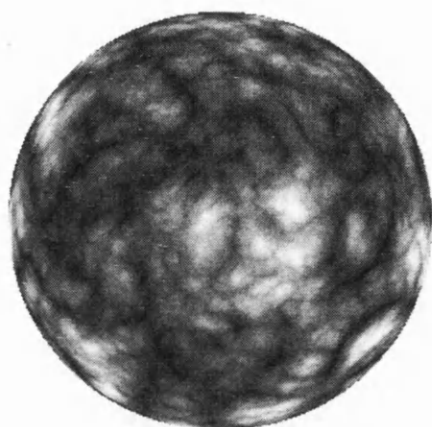
The performance and visual results of these locally evaluated noise and higher order noise techniques are discussed in section 4.1.3 where a comparison to pre-computed noise blocks is presented.

4.1.3 Results

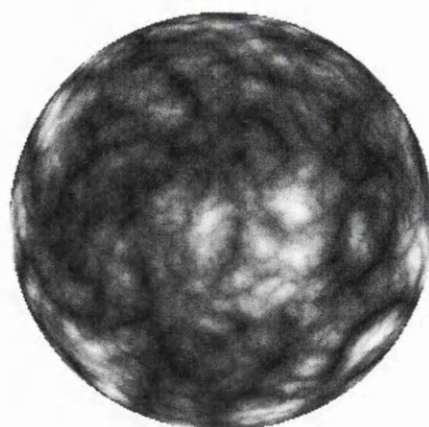
The timings and quality assessment of each GPU noise implementation is based on a simple solid texturing of a volumetric object (see section 4.2). The OOP method for iso-surfacing slice samples is adapted to include the solid texturing stage of the volume pipeline. No rotations, lighting or optimisations are applied to the volume in order to differentiate each noise methods performance. The performance measurements are taken in this manner since simply applying a noise implementation to a simple surface geometry will not gain any insight because few fragments are considered and additionally the problem of volume rendering defines the base overhead. The dataset used is the *SphereDist* dataset, which is 256^3 dimensions and contains 32 bit floating point data. There are 256 sample proxy slices considered during volume rendering.

Table 4.1 shows the frame rates for each different noise implementation and higher order noise primitives. The lookups column represents the number of texture lookups made in the fragment shader to provide the noise primitive. The shader mode is set to take conditional branches in favour of condition code execution such that noise algorithms are not computed for non iso-surface samples. The OOP approach however does not contain any empty space leaping or early ray termination speed-ups, therefore every sample encountered that is greater than the iso-value threshold is subject to noise computation. The iso-surfacing shader without noise is set to use conditional branching also to provide a direct comparison.

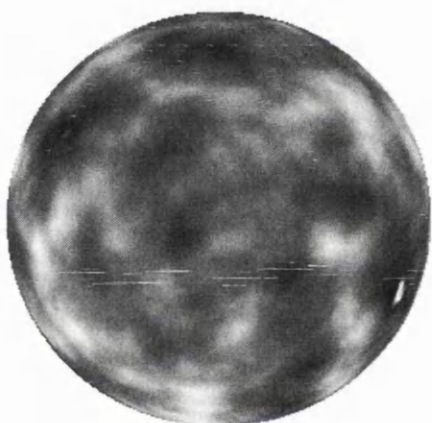
Figure 4.2 and 4.3 shows each noise implementations visual result. Locally evaluated noise produces the best visual results over any resolution or viewport due to the cubic interpolation



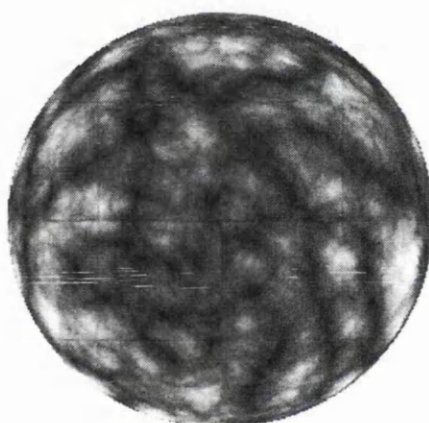
(a) Locally Evaluated - 8 Octaves, 72 Lookups



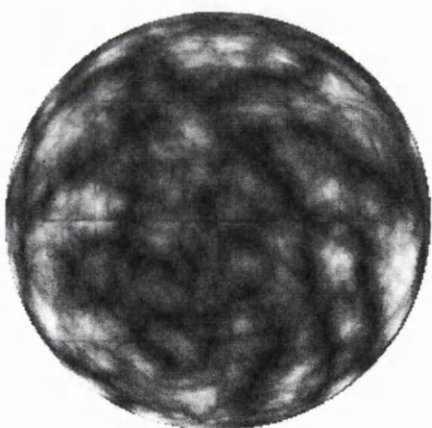
(b) Locally Evaluated - 4 Octaves, 36 Lookups



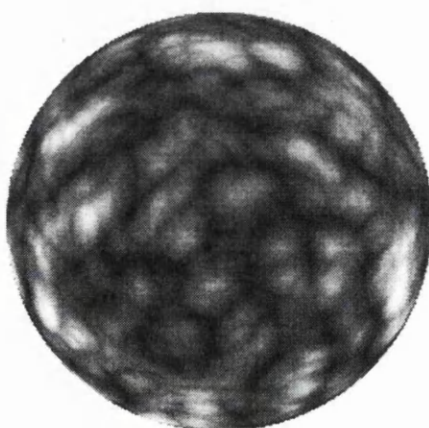
(c) Pre-computed - 8 Octaves, 1 Lookup



(d) Pre-computed - 8 Octaves, 8 Lookups



(e) Pre-computed - 4 Octaves, 4 Lookups



(f) Pre-computed - 4 Octaves, 1 Lookup

Figure 4.3: Examples of turbulence implementations

Viewport	Method	Primitive	Octaves	Lookups	FPS
512 ²	Iso-surfacing	n/a	n/a	n/a	38
	Locally evaluated (see section 4.1.2)	Noise	1	9	7
		Turbulence	4	36	2
		Turbulence	8	72	1
	Pre-computed (see section 4.1.1)	Noise	1	1	35
		Turbulence	4	1	24
		Turbulence	4	4	11
		Turbulence	8	8	6
		Turbulence	8	1	35
1024 ²	Iso-surfacing	n/a	n/a	n/a	18
	Locally Evaluated (see section 4.1.2)	Noise	1	9	2
		Turbulence	4	36	< 1
		Turbulence	8	72	< 1
	Pre-computed (see section 4.1.1)	Noise	1	1	14
		Turbulence	4	1	8
		Turbulence	4	4	4
		Turbulence	8	8	1
		Turbulence	8	1	14

Table 4.1: GPU Noise implementation comparisons. All FPS are rounded down

carried out for each sample location. Locally evaluated noise within the fragment shader is shown to be expensive to compute and is only a viable option for efficient algorithms where minimal noise function calls are required such as deferred shading techniques.

The pre-computed noise block implementation will interpolate linearly between known texels in the pre-computed noise block. Therefore a smoothing of the noise function is evident if the pre-computed noise block is not large enough to have the same resolution as a locally evaluated counterpart. The turbulence tests for pre-computed noise blocks using multiple lookups produce artifacts when addressing outside the texture block's domain. This problem can be avoided by increasing the texture's domain to include all values that will be addressed when building the turbulence function. However this implies a larger texture size to be pre-computed and sampled. This method does remove the problem of magnification smoothing the noise signal and produces better visual results over differing resolutions.

Building turbulence from pre-computed noise blocks is possible given the correctly computed domain, although a pre-computed turbulence block produces the best results for a fixed resolution. The implementation with textures that encodes differing noise frequencies in each channel does not suffer from bordering issues and provides the best GPU memory consumption and visual quality mechanism to provide noise primitives for general computation. This technique gives the ability to have 4 differing frequencies of noise and additionally the ability to build higher order noise primitives such as turbulence with good visual quality and performance. This technique also improves on texture magnification smoothing the noise signal.

Pre-computed noise blocks are therefore the fastest noise implementation available for GPU

volume graphics pipelines and locally evaluated noise functions in the fragment shader prove to be available for algorithms exhibiting deferred shading. The complexity of a noise implementation is an important factor in procedural volume rendering techniques and a noise implementation that scales well over the volume rendering problem is important to describe the algorithms general characteristics and runtime performance. It can also be seen that for the size of viewport and volume dataset being considered for interactive techniques, a 4 octave implementation of turbulence produces very similar to an 8 octave implementation. The remainder of this chapter focuses on pre-computed texture blocks in order to balance the overall volume rendering pipeline and implement the most efficient methods.

4.2 Solid Texturing Volumetric Objects

This section introduces interactive procedurally evaluated solid texturing of volumetric objects to the GPU. Solid texturing or carving is particularly useful for modelling naturally occurring materials such as wood and marble among many others. This approach adapts best with the volume graphics paradigm since a 3D domain is considered to render a volume dataset. Additionally volume datasets are capable of representing gaseous phenomena such as fire and smoke. Therefore texturing with volume datasets presents the best approach to describing a volume graphics pipeline. In addition solid texturing or volume texturing is the most intuitive method for synthesising objects from solid materials. Texture mapping a 3D primitive is achieved by looking up the position in 3D space to be textured with the corresponding location in a volume texture map or texture function (see Eqn 4.3)

$$\begin{aligned} m : \mathbf{P}^3 &\rightarrow \mathbf{C} \\ t &= m(x, y, z) \end{aligned} \tag{4.3}$$

where $\mathbf{P}^3 \in \{0, \dots, 1\}^3$ is used to represent a 3D point in unit texture space. t represents the resulting four channel texel, where $\mathbf{C} \in \{0, \dots, 1\}^4$ to represent an $\langle RGB\alpha \rangle$ quadruple.

Solid texturing volume objects is achieved by iso-surfacing a volume dataset and fetching a solid texture texel for each iso-surface location. This is then generally subject to lighting calculations if the texture map returns colour information or classification and lighting if a scalar is returned. The most simple solid texturing method involves simply using the iso-surfaces object space coordinates to perform a lookup into the solid texture map. To provide a mechanism to allow for arbitrary procedural solid texturing, this simple method is expanded to include procedural texturing primitives and a dynamic shading extension to the fragment shader to allow arbitrary procedural textures to be specified and generated on the fly. This is achieved by providing a noise implementation and additionally providing a function call mechanism to allow shading language libraries to define functions for a shader at runtime. The shader will then be compiled whilst the application is running and uploaded to the GPU. Functions are inlined during on the fly compilation. The noise implementations are discussed in section 4.1.

To successfully allow for solid texture space transformations the vertex shader outputs two sets of texturing coordinates for volume dataset object space coordinate lookups. One of these sets of co-ordinates defines the sampling locations for volume rendering whilst the

other defines solid texturing coordinates that are defined from the original object space coordinates. These additional coordinates can be transformed in the vertex shader with a transformation matrix in order to ensure the volumes object space coordinates and solid texture coordinates can be disjoint. This transformation is applied in the vertex shader to remove the per fragment overhead. The texture coordinates are linearly interpolated across any proxy geometry during rasterization. Both slice and slab sampling methods are implemented for OOP and IOS techniques. The defined mechanism of providing ray starting locations and direction vectors for orthographic projections in image-order techniques requires that the solid texture transforms are applied in the fragment shader since an additional bounding geometry would have to be used in additional passes to correctly define a disjoint co-ordinate set. This restriction only applies to the mechanism where two initial passes are employed to encode a volume bounding box. Other schemes of perspective projection and orthographic projection compute ray starting locations and step vectors in the fragment shader from rasterized input obtained from a single quadrilateral.

The fragment shaders detailed use two methods *snoise* and *sturbulence* which are provided by pre-computed noise texture blocks or locally evaluated functions as discussed in section 4.1. These implementation specific details for each noise method are omitted for clarity. Most GPU shading languages have a reserved function *noise* for future noise algorithm hardware. Currently GPU implementations return 0 or 1 and in some cases return a value for noise which is based on the same mechanism outlined for locally evaluated noise. There is currently no implementation that uses a specialised hardware noise unit². The description of each method will contain a function *solidTexture(pos)* which will take the solid texture co-ordinates to compute a new surface scalar that will be subject to classification with a transfer function. This mechanism is preferred to including a transfer function to define an environment where texture is always applied in the same manner.

Marble is defined using the *turbulence* function and the periodic function *sin* to produce the wavy line pattern (see Eqn 4.4). A colour spline is used to classify the result of the adjusted *turbulence* function and provide the marble colouring. One dimension of the domain is offset to allow marble density to appear more compact in one particular direction.

$$marble(s, t, r) = \sin(s + turbulence(s, t, r)) \quad (4.4)$$

where $s, t, r \in \mathbb{R}$ define the 3D solid texture co-ordinate.

Wood is defined using the *noise* function. A quadratic equation is used to yield a concentric set of cylinders (see Eqn 4.5). Noise is used to add waviness to the grain. The quadratic characteristic makes the early grain appear wider than later grain which is consistent with tree development. The wood transfer function can be simply implemented with a linear interpolation between dark brown and light brown.

$$wood(s, t, r) = s^2 + t^2 + noise(4s, 4t, r) \quad (4.5)$$

where $s, t, r \in \mathbb{R}$ define the 3D solid texture co-ordinate.

²Perlin maintains a patent on a hardware implementation of the Perlin noise algorithm which is expected to appear on GPU hardware in the future

```

fragment vertexShader(vertex, modelViewMatrix, textureMatrix, solidMatrix)
    fragment.pos = vertex.pos * modelViewMatrix
    fragment.tex0 = vertex.tex0 * textureMatrix
    fragment.tex1 = vertex.tex1 * textureMatrix
    fragment.tex2 = vertex.pos * textureMatrix.inverseTranspose
    fragment.tex3 = vertex.tex0 * solidMatrix;
    fragment.tex4 = vertex.tex1 * solidMatrix;

```

Figure 4.4: OOP solid texturing vertex shader

```

pixel fragmentShader(fragment, volume, transfer, isoValue, light, textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue.a)
        light.diffuse = transfer(solidTexture(IN.tex3))
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif

```

Figure 4.5: OOP solid texturing fragment shader

4.2.1 Object-Order Proxy Slice Solid Texturing

The OOP rendering method for iso-surfaces (see section 3.4.1) is adapted to solid texturing by including an additional set of texture coordinate outputs from the vertex shader to the fragment shader (see Figures 3.20 and 4.4). The iso-surfacing fragment shader for slices (see Figure 3.25) is adapted to include a solid texturing stage after segmentation and before classification and lighting operations (see Figure 4.5).

To consider slab sampling over slice sampling, the interpolated multiple iso-surfacing technique (see section 3.3.3) is adapted to solid texturing by replacing the colour classification stage. Two texture maps are uploaded to the GPU in multiple iso-surface techniques, one for the colour classification and the other to hold the interpolation weights. Since the solid texturing mechanism preceeds the colour classification stage, the colour texture can be omitted from upload and the pre-processing step reduced to only calculate interpolation weights. The fragment shader for interpolated multiple iso-surface rendering (see Figure 3.27) is adapted to alter the classification stage (see Figure 4.6). The interpolation scheme interpolates the position to be exactly on the iso-surface with the weighting table. In this manner only one solid texturing routine is required to evaluate an interpolated position. Visual artifacts are introduced by interpolating the results of two solid texturing calls at each sample point. This is due to the high-frequencies evaluated during noise calculations where an interpolated scalar can be derived from differing scalars from the front and back sample positions and the result will be an erroneous linear mix.

This method includes the ability to define multiple iso-surfaces to solid texture within a volume dataset. This is possible by altering the solid texture function to allow a further classification variable to be included. Each iso-surface is assigned a different colour within the original multiple iso-surface rendering method which is used to segment each iso-surface. This process involves uploading the colour table along with the original weighting table.


```

pixel fragmentShader(fragment, volume, transfer, weight, light, textureMatrix)
    voxelF = volume(fragment.tex0)
    voxelB = volume(fragment.tex1)
    wght = weight(voxelF.a, voxelB.a)
    if (wght > 0.0)
        pos = lerp(IN.tex3, IN.tex4, wght);
        normal = (lerp(voxelF.xyz, voxelB.a, wght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        light.diffuse = transfer(solidTexture(pos))
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif

```

Figure 4.6: OOP interpolated solid texturing fragment shader

```

pixel fragmentShader(fragment, volume, dir, isoValue, light, transfer,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(rayPos)
        if (voxel.a > isoValue.a)
            break
        endif
        rayPos += direction
        .
        // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (voxel.a > isoValue.a)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        light.diffuse = transfer(solidTexture(rayPos))
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 4.7: IOS solid texturing fragment shader

4.2.2 Image-Order Single Pass Solid Texturing

The IOS method is adapted to solid texturing in the same manner as OOP methods. Figure 4.7 is the modified fragment shader of figure 3.41 to include solid texturing for the IOS method for slice sampling iso-surfaces. Figure 4.8 is the modified fragment shader of figure 3.42 to include solid texturing for the IOS method with interpolated multiple-iso surface sampling. Only one iso-surface is considered for solid texturing, however multiple iso-surfaces are possible analogous to the OOP methods described in section 4.2.1.

Whilst the complexity of the underlying rendering problem remains the same, additional speed-ups can be computed since the whole ray is considered in one fragment shader. This gives rise to using dataset dependant information to encode strategies that reduce the number of samples required. Empty space leaping can be performed as outlined in section 3.2.4 for distance field datasets. The ray increment presented in the fragment shaders in this section are considered to be replaced with the distance field stepping function. Octree structures can also be used to allow a small speed-up. This small speed-up is due to skipping


```

pixel fragmentShader(fragment, volume, dir, isoValue, light, transfer,
    textureMatrix, weight)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    voxelF = volume(rayPos)
    rayPos += direction
    while (true)
        voxelB = volume(rayPos)
        wght = weight(voxelF.a, voxelB.a).a
        if (wght > 0.0)
            break
        endif
        voxelF = voxelB
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (wght > 0.0)
        normal = (lerp(voxelF.xyz, voxelB.xyz, wght) * 2.0 - 1.0)
        * textureMatrix.inverseTranspose
        rayPos = lerp(rayPos - direction, rayPos, wght)
        light.diffuse = transfer(solidTexture(rayPos))
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 4.8: IOS *interpolated solid texturing fragment shader*

costly instructions in the fragment shader during ray traversal, however every sample must be visited since a given volume orientation does not allow skipping over the empty space without sampling because the distance to each octree cell is unknown without expensive ray intersection equations. In general the octree mechanism is better suited to object-order approaches. Distance fields can also be used to perform empty space leaping along a ray and are better suited to the image-order approach since the distance in voxels to the closest point of the iso-surface being rendered is encoded instead of the scalar field. Additional speed-ups include early ray termination and additionally deferred shading which allows the application of solid texture once thought for each ray that intersects the iso-surface. This deferred shading strategy also benefits from not having to compute dynamic branches inside the ray sampling loop as the instructions required are significantly reduced. The combined speed-up mechanisms of empty space leaping, early ray termination and deferred shading when compared on the same architecture and rendering strategy produce 43 fps instead of 14 fps when no speed-up's are included on the *CTHeadDist* dataset for 256 samples along the ray. In general these speed-ups are dependant on the dataset being rendered and produce differing levels of acceleration for different iso-surfaces or fuzzy classification tasks.

4.2.3 Results

Two algorithms are explored for solid texturing. OOP rendering and IOS rendering. The two adaptations of the original volume iso-surface algorithms are outlined in sections 4.2.1 and 4.2.2 respectively. Noise and turbulence are implemented for the results in the fastest

manner available (see section 4.1). Timings are taken with no volume rotation to provide a performance comparison that is not affected by cache misses in 3D texturing hardware and to ensure the amount of fragments being processed are uniform throughout the tests.

Three different procedural textures are tested to allow the performance overhead of the approach to be disjoint from the particular textures overhead. The wood solid texture is used for demonstration of quality purposes since the more noisy functions such as noise, turbulence and marble mask under-sampling due to their highly stochastic properties. Wood is a more regular pattern that requires a higher level of precision to accurately reproduce the grain. Turbulence is used to demonstrate a complex and detailed procedural texture and Perlin's bozo [Per85] texture is used to demonstrate both of these properties. The *BuckyBall* dataset is encoded with the bozo texture in order to act as a comparison to previous timings and image quality analysis with the same iso-value. The *CTHeadDist* distance field dataset is also explored to demonstrate the increased rendering speed possible when using an efficient empty space leaping method for image-order approaches. The *SphereDist* dataset is used because it is a regular shape with uniformly different gradients over its surface and can highlight under-sampling in lighting calculations.

Noise (or turbulence) is the fastest method considered as this can be achieved with a single texture lookup. Faster procedural routines are possible without the use of noise, however noise or turbulence is predominantly used in procedural modelling and is considered here to be a fundamental building block. Bozo and wood are demonstrated as more complex procedurally generated solid textures. These two synthesised patterns require a texture lookup into the noise texture and additionally a texture lookup for colour classification as well as some instructions which are processed in the fragment shader.

Table 4.2 shows the frame rates achieved for solid texturing with the OOP rendering method. Timings are taken without any lighting calculations to demonstrate the maximum throughput of each technique. The images obtained for each test are presented in figures 4.9 - 4.11. These figures each depict a different procedural solid texture and each image contains lit and unlit solid textured surfaces for quality comparison. Noisy textures hide under-sampling artifacts in most cases and lower precision rendering can be employed for a good aesthetic representation. The slab rendering algorithm gives significantly better visual results for the same samples taken along a ray in comparison to the slice technique. Images in figures 4.9(b) and 4.9(h), 4.10(a) and 4.10(g), 4.11(b) and 4.11(h) are comparable in terms of image quality where there are little to no artefacts included in the output due to under-sampling and there is additionally no noticeable aliasing of the texture, however the slab method of sampling yields better performance than the slice method for a similar quality representation.

Table 4.3 shows the frame rates achieved with the IOS method, computing each ray entirely in one pass in the fragment shader. The difference in performance for each sampled solid texture function is negligible. The results are slower than the OOP approach, however this approach includes early ray termination and deferred shading. This approach is therefore more scalable than the OOP approach when rendering into larger viewports and increasing sampling frequency due to less samples being visited. Additionally shading is performed only once per ray in a deferred manner which allows for more complex lighting and solid texturing techniques to be realised in real time. The deferred shading of the iso-surface does

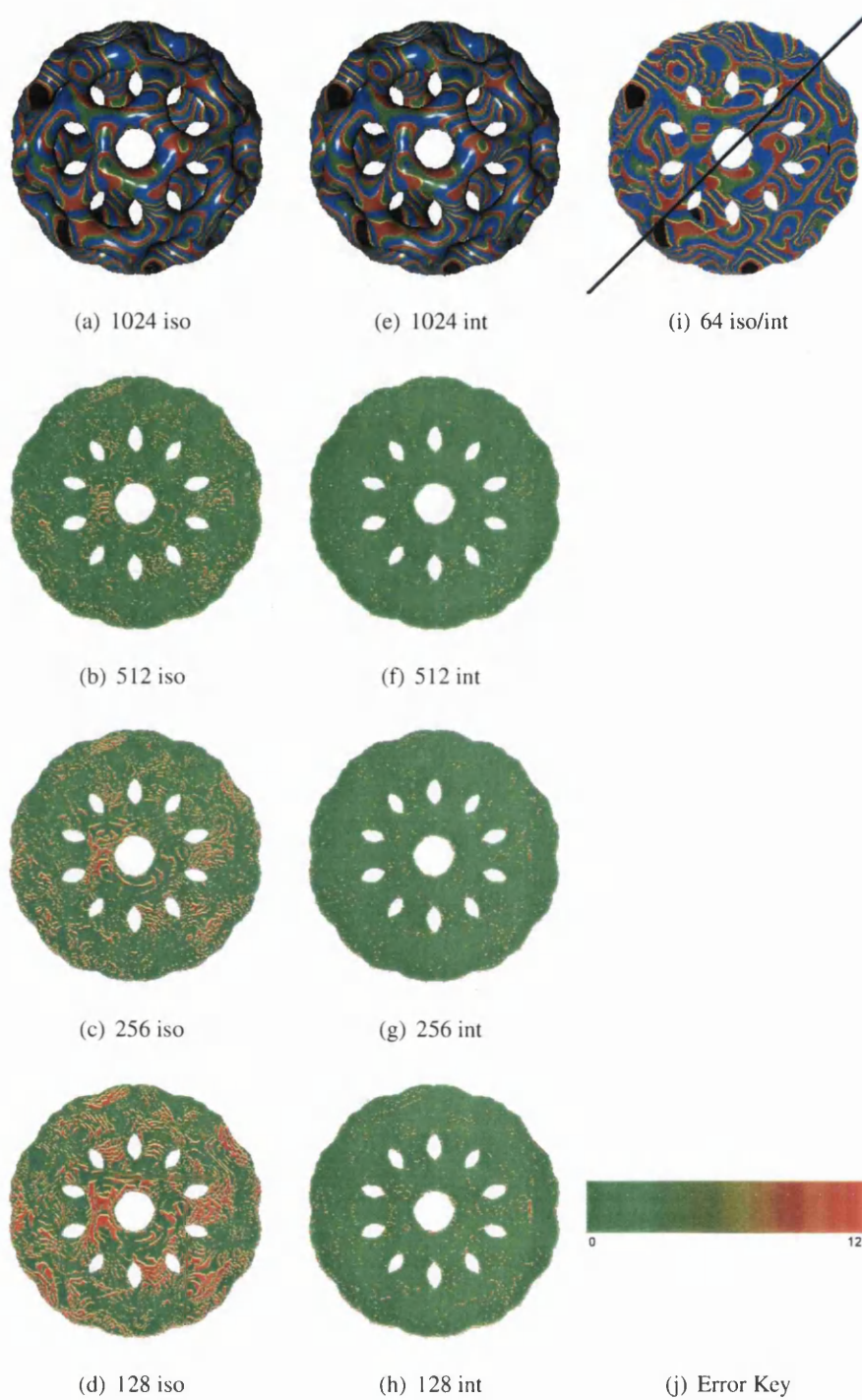


Figure 4.9: BuckyBall dataset bonzo solid texture (a) to (d) and interpolated solid texture (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared with no shading contributions to highlight sampling differences and the error range for difference images is given in (j).

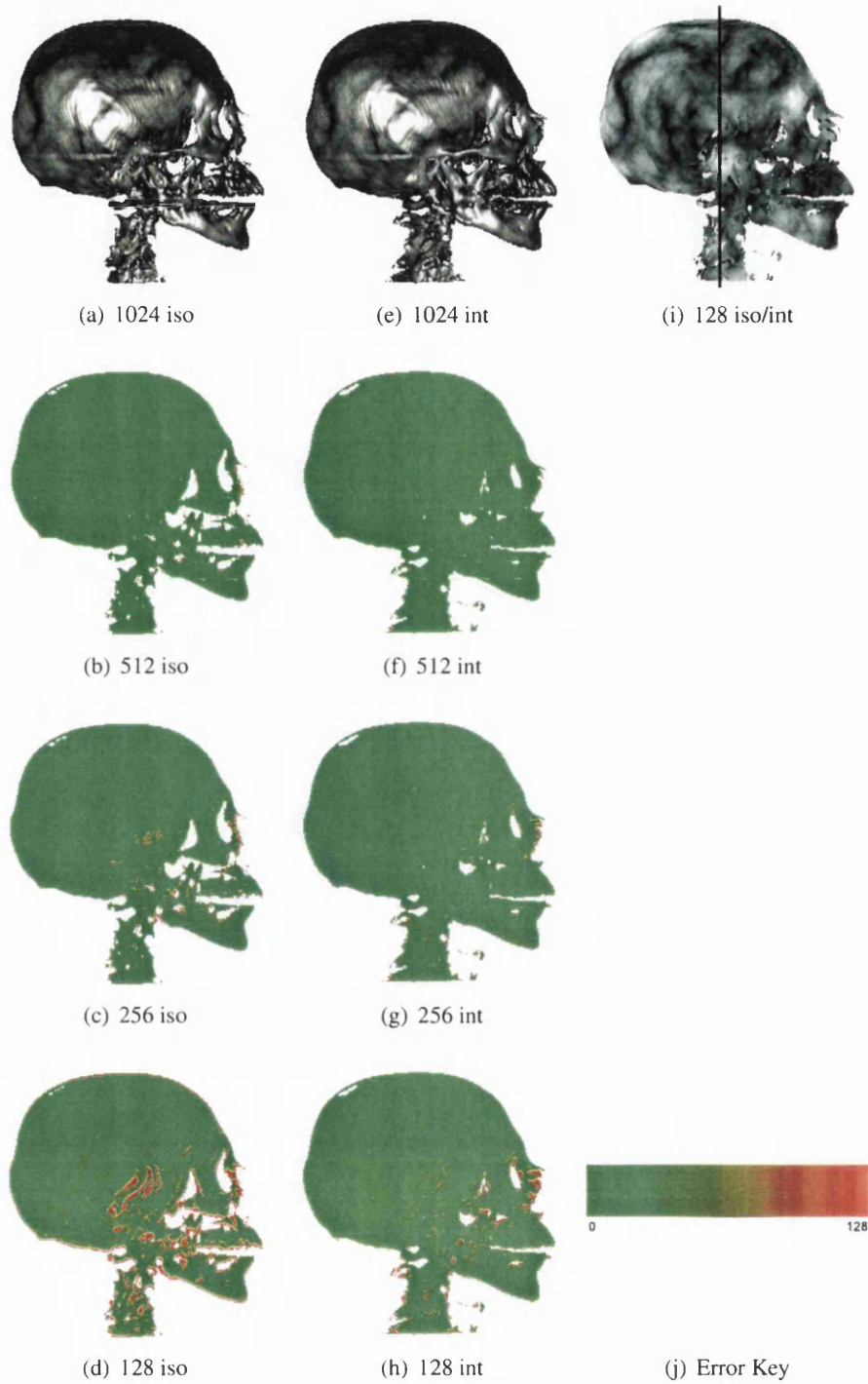


Figure 4.10: *CTHeadDist* dataset turbulence solid texture (a) to (d) and interpolated solid texture (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared with no shading contributions to highlight sampling differences and the error range for difference images is given in (j).



Figure 4.11: *SphereDist* dataset wood solid texture (a) to (d) and interpolated solid texture (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates.. Both techniques (i) are compared with no shading contributions to highlight sampling differences and the error range for difference images is given in (j).

Dataset (size)	Viewport	Slices	Noise		Wood		Marble	
			Iso	Int	Iso	Int	Iso	Int
<i>BuckyBall</i> (32 ³) see Figure 4.9	512 ²	128	92	52	71	41	90	50
		256	48	28	38	22	49	27
		512	26	14	20	10	26	13
		1024	14	7	10	5	13	7
	1024 ²	128	29	14	20	11	28	14
		256	14	7	10	5	14	7
		512	7	3	5	2	7	3
		1024	3	1	2	1	3	1
<i>CTHeadDist</i> (256 ² × 128) see Figure 4.10	512 ²	128	66	34	53	29	55	30
		256	33	16	26	13	28	15
		512	17	8	13	7	14	6
		1024	8	4	6	3	7	3
	1024 ²	128	28	15	20	11	27	14
		256	14	7	10	5	14	7
		512	7	3	5	2	7	3
		1024	3	1	2	1	3	1
<i>SphereDist</i> (256 ³) see Figure 4.11	512 ²	128	62	30	50	26	59	35
		256	33	17	26	14	36	15
		512	17	8	13	6	16	8
		1024	8	4	6	3	8	4
	1024 ²	128	27	14	20	11	28	14
		256	14	7	10	5	14	7
		512	7	3	5	2	7	3
		1024	3	1	2	1	3	1

Table 4.2: OOP solid texturing frame rates in frames per second, Iso is single sample solid texturing and Int is interpolated solid texturing. All rates are rounded down.

Dataset (size)	Viewport	Slices	Iso	Int
<i>BuckyBall</i> (32^3) see Figure 4.9	512^2	128	40	27
		256	21	13
		512	12	7
		1024	5	3
	1024^2	128	20	14
		256	10	7
		512	5	3
		1024	2	1
<i>CTHeadDist</i> ($256^2 \times 128$) see Figure 4.10	512^2	128	29	21
		256	14	10
		512	7	5
		1024	3	2
	1024^2	128	10	6
		256	5	3
		512	2	1
		1024	1	< 1
<i>SphereDist</i> (256^3) see Figure 4.11	512^2	128	26	19
		256	13	9
		512	6	4
		1024	3	2
	1024^2	128	9	6
		256	4	3
		512	2	1
		1024	1	< 1

Table 4.3: IOS solid texturing frame rates in frames per second, Iso is single sample solid texturing and Int is interpolated solid texturing. All rates are rounded down.

allow this approach to use locally evaluated noise functions at interactive rates.

Table 4.4 shows the results for rendering distance fields with early ray termination and empty space leaping with deferred shading. The performance increase over the previous results for the IOS approach is due to allowing large areas of the volume that do not contribute to the surface to be successfully skipped (see section 3.2.4). An empty space leaping value table is pre-computed in these measurements and proved to outweigh a fragment shader function due to one texture lookup being required instead of multiple instructions to compute the space leaping function (see section 3.2.4). Adaptive step sizes are used since a given distance leap vector is compared in length to the step size vectors length and the maximum is always used. This allows adaptive rendering of important features and concentrates sampling where most required. In practice the rendered results contain less artefacts due to concentration on required areas of the volume dataset. This makes the space leaping mechanism require consideration of the current step size as the approach of skipping up to a 1 voxel border around the iso-surface is less efficient when step sizes are large. Therefore the performance characteristics of this approach make it very scalable for differing step sizes and viewports due to less samples being taken along each ray, which is dependent on dataset construction and the desired final output region of interest. In general, most datasets such as the *CTHeadDist* can benefit greatly from these accelerations due to empty space surrounding the regions of interest in the dataset and thus several samples along a ray can be omitted. This property is the result of uniformly skipping empty space for arbitrary step sizes and benefits from large step sizes. Therefore the image-order approach with empty space skipping using distance fields, deferred shading and early ray termination gives the best overall approach to a scalable solid texturing environment and includes the ability to use locally sampled procedural primitives at interactive rates.

Dataset (size)	Viewport	Slices	Iso	Int
<i>CTHeadDist</i> ($256^2 \times 128$) see Figure 4.10	512 ²	128	56	34
		256	43	25
		512	30	20
		1024	18	14
	1024 ²	128	19	16
		256	15	11
		512	12	9
		1024	9	5

Table 4.4: IOS solid texturing frame rates with empty space leaping in frames per second, Iso is single sample solid texturing and Int is interpolated solid texturing. All rates are rounded down.

4.3 Volume Hypertexture

Hypertexturing complex volumetric primitives is achieved by adapting Perlin and Hoffet's [PH89] original object density implementation for implicit surfaces (see Eqn 2.26) to compute over a volume dataset. Satherley and Jones [SJ02] introduce an object density function

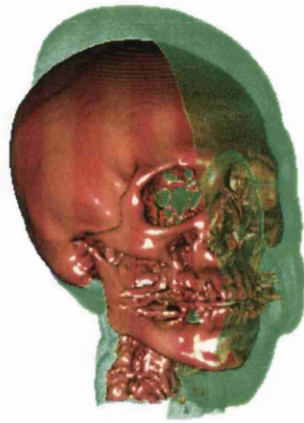


Figure 4.12: *CTHeadDist* dataset with object density function defining a soft-region. The soft region is clipped to show the relationship to its object

for volume datasets using distance fields. This section introduces hypertexture for volumetric primitives on GPU hardware and defines an adaptable procedural hypertexture pipeline which achieves real-time frame rates.

Hypertexturing is an important texturing technique in the volume graphics pipeline since amorphous and gaseous phenomena and naturally occurring object properties can be generated with this approach. These models are not possible intuitively with surface based graphics since each surface is treated as infinitely thin. Therefore less intuitive and more complex techniques must be used in surface based graphics to import naturally occurring object properties.

Distance field datasets are used because a segmentation of the object is required to allow the definition of a soft-region, or malleable region around the object's surface. A distance field is especially suited to this since the distance values contained in the dataset can be directly utilised for this segmentation. A standard volume dataset containing density scalars would require pre-processing into a distance field.

4.3.1 Distance Fields

Distance fields are generated from many sources such as triangle meshes, implicit functions and volume datasets to contain an iso-surface of interest. Section 3.2.4 outlines the methods for GPU rendering of iso-surfaces contained in distance field volumes. This involves making a binary segmentation during fragment shading to correctly identify the iso-surface. For hypertexture effects a soft-region is required and the classification stage is altered to perform two segmentations. Two iso-values are thus required, one for the original surface (usually 0) and one for the desired soft-region. This is expressed in terms of voxels since the distance function contains Euclidean distances. This is achieved by replacing the binary segmentation for iso-surface rendering with an object density function for distance field volume datasets. This allows the classification of the distance function into three disjoint

sets.

- Outside the object
- The soft-region of the object's surface
- Inside the object

The object density function for distance fields is defined in Eqn 4.6. Figure 4.12 shows an example of the object density function applied to the *CTHeadDist* dataset to form a soft-region. This soft-region has been clipped to show differences in detail. The object density function is examined at run-time with DMF functions being applied to the segmented soft-region. Perlin's [PH89] original work does not describe rendering the surface of an implicit object at its surface (when $D(p) = 1$). This hypertexture implementation also covers rendering the iso-surface contained in the distance field as this feature is required for many hypertexture effects.

$$D(p) = \begin{cases} 1 & \text{if } |p| \leq r_i^2 \\ 0 & \text{if } |p| \geq r_o^2 \\ \frac{|p| - r_o}{r_i - r_o} & \text{otherwise.} \end{cases} \quad (4.6)$$

where r_i is the inner distance or iso-surface, r_o is the outer distance or soft-region boundary and $|x|$ represents the distance field value [SJ02]. 1 is returned for samples inside the object and are subject to iso-surfacing, 0 is returned for samples outside the object and soft-region boundary and $\frac{|p| - r_o}{r_i - r_o}$ the distance from the surface is returned when samples are between the soft-region boundary and surface boundary.

4.3.2 DMF Functions

The object density function is applied during fragment shading of a sample. DMF functions are applied to soft-region samples and involve compositing of samples analogous to fuzzy segmentation techniques. Therefore a mixed mode rendering strategy is required to correctly compute a hypertexture effect. A subset of possible DMF functions can be pre-computed, loaded into GPU memory as a volume texture and addressed with the corresponding sampling coordinates. However the goal is to provide a platform for interactive procedural hypertexturing and arbitrary DMF functions are defined in the fragment shader and computed on the fly. This has an effect rendering performance depending on the amount of extra instructions the fragment shader must process. Generally noise is used as a primitive for procedural techniques and must be present to the fragment shader as outlined in section 4.1. Noise is considered a fundamental building block for procedural techniques, although effects can be produced without its use. Attributes available for computing DMF functions include:

- 3D object position
- Current volume distance value
- Current samples gradient normal

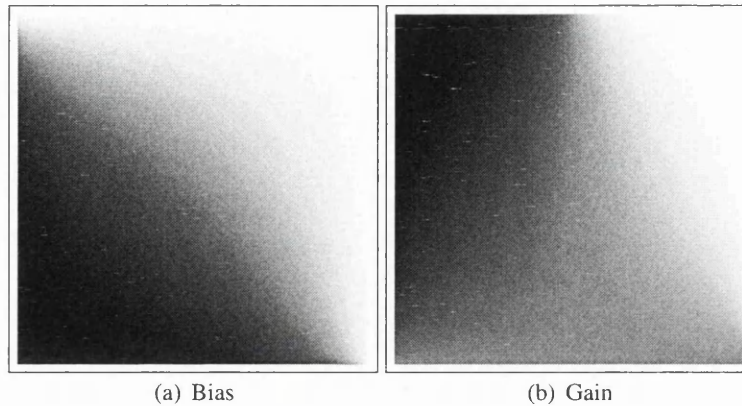


Figure 4.13: *Bias and Gain lookup tables*

- Noise function
- Noise gradient normals (texture based noise)
- Standard functions on GPU such as floor, ceil, clamp, sin, cos
- *Arbitrary transfer functions encoded as textures
- *Arbitrary uniform variables (floats, vectors and matrices defined for the fragment shader)

* items are not included in the description of the hypertexturing pipeline presented as most hypertexture effects do not require additional parameters. One transfer function texture is provided to allow colouring of densities generated by DMF functions. In addition fast texture based lookup tables for the bias and gain functions are provided (see Figure 4.13).

A further segmentation of the soft-region allows different DMF functions to be chosen through the soft-region based upon parameters such as spatial co-ordinates or distance from the surface. This implementation does not describe segmenting the soft-region, however it can be achieved by building a DMF function to perform the segmentation. Additional operations such as lighting can also be included in DMF functions since the parameters are available to the fragment shader. A generalised shader model is presented since using high level GPU shader languages, functions can be defined and linked at runtime dynamically and swapped in or out on the fly. This technique allows implementation of a flexible hypertexturing modelling environment for volumetric primitives.

Each fragment shader will call a function *hypertexture(pos, normal, density, volume)* which takes each parameter required for application of DMF functions. DMF functions can be built within this function and a scalar will be returned for classification. A further shade trees [Coo84] implementation enables changes in lighting and classification parameters as a post-processing step. Lighting calculations require that each sample computes central differences (see section 2.4) in respect of the hypertexture function for a correct gradient to be derived. Pre-computing these gradients is not possible since the hypertexture DMF functions are computed on the fly. Therefore any fragment shader that is to calculate lighting

is required to perform three additional hypertexture operations per sample.

4.3.3 Example Hypertextures

The following equations define DMF functions to compute hypertexture. Equation 4.7 introduces electric storm, a position dependent function which enables a pseudo lightening effect to be created. Turbulence is used as a base DMF function and is further displaced with the *sin* function. Figure 4.14(h) is an example image rendered with this function.

$$electricStorm(D(p), p) = \sin(turbulence(f p)) \quad (4.7)$$

where f is the desired noise frequency.

Equation 4.8 demonstrates how melting [PH89] is achieved by displacing a component of the incoming sample position. This requires a further lookup into the volume to assess the density discovered at the new sample position and is thus a position dependent function (see Figure 4.14(g)).

$$melt(D(p), p) = D(p_x (1.0 + noise(p))) \quad (4.8)$$

where p_x represents the x component of the sample position p , p_y and p_z can also be used. This function can also alter the direction of melting by changing $1.0 + noise(p)$ to $1.0 - noise(p)$.

Fur and curly fur [PH89] are modelled by an approximation of a hair filament being grown from the objects surface. This requires that the inverse of the gradient is projected onto the objects surface and is therefore a geometry dependent function. Noise is evaluated at the objects surface and is used throughout the width of the soft-region. This allows straight hair filaments to be built up into a fur effect (see Eqn 4.9 and Figure 4.14(e)). Bias and gain are used to control the number of hair filaments and the boundary respectively. The constant values are suggestions from Perlin's [PH89] original work.

$$fur(D(p), p) = gain_{0.9}(bias_{0.3}(noise(f project(p)))) D(p) \quad (4.9)$$

where f is the desired noise frequency.

Curly fur follows the same procedure as straight fur and is modelled by offsetting the projected sample on the surface uniformly throughout the width of the soft-region. This is achieved by offsetting the sample position before projection, allowing a differing noise value to be encountered at the surface.(see Eqn 4.10). An offset gradient is introduced for this purpose and is controlled with the density encountered for the current sample. In this manner the amount of curl is controllable. The additional gain function call allows shaping of the hair filament (see Figure 4.14(f)).

$$\vec{noise} = \langle noise(p - \sigma), noise(p), noise(p + \sigma) \rangle \quad (4.10)$$

$$\text{curlyFur}(D(p), p) = \text{gain}_{0.9}(\text{bias}_{0.3}(\text{noise}(f \text{ project}(p')))) D(p)$$

where $p' = p + \text{gain}_{0.8}(1.0 - D(p)) \text{curl noise}(p)$ and f is the desired noise frequency.

Fire can be implemented in differing manners. Perlin [PH89] originally defined fire effects by displacing the current sample position with noise for a further lookup into volume densities (see Eqn 4.11 and Figure 4.14(a)). This method is expensive to compute in real-time since an additional texture lookup is required into the volume dataset.

$$\text{perlinFire}(D(p), p) = D(p (1.0 + \text{turbulence}(p))) \quad (4.11)$$

The following functions are introduced as alternative or approximate fire effects. Fireball is defined by using turbulence as a term to alter the soft-region density field. This method respects the density distribution through the width of the soft-region and avoids the extra texture lookup. A simpler fire effect can be implemented with no regard to the soft-region density distribution (see Eqn 4.13 and Figure 4.14(c)). A simple extension to this function to allow greater realism includes a post DMF computation (see Eqn 4.14 and Figure 4.14(d)). These additional functions reduce the complexity of the original implementation and provide more efficient alternatives.

$$\text{fireball}(D(p), p) = D(p) (1.0 + \text{turbulence}(p)) \quad (4.12)$$

$$\text{fire}(D(p), p) = \text{turbulence}(p) \quad (4.13)$$

$$\text{lumpyFire}(\text{lut}, D(p), p)_{rgb} = \text{lut}(\text{fire}(D(p), p))_{rgb} \quad (4.14)$$

$$\text{lumpyFire}(\text{lut}, D(p), p)_a = (\text{lut}(\text{fire}(D(p), p))_a - k) - \text{fire}(D(p), p)$$

where k is a control constant for lump severity.

Bias and gain hooks are included into DMF functions to allow fine control of the distribution of density through the soft-region. Additionally the frequency of noise is a useful control along with the transfer function to map colour and opacity to densities in the soft-region.

4.3.4 Pre-Integrated Transfer Functions

Hypertexture effects require a fuzzy classification of the soft-region in order to correctly colour the DMF function. Using Pre-integrated 3.3.3 transfer functions and the slab style of rendering can produce less artifacts in the soft-region and reduce under-sampling. Slabs are considered instead of slices which involves two lookups along the ray to define one sampling position. Therefore, since the densities encountered are to be composited, two hypertexture

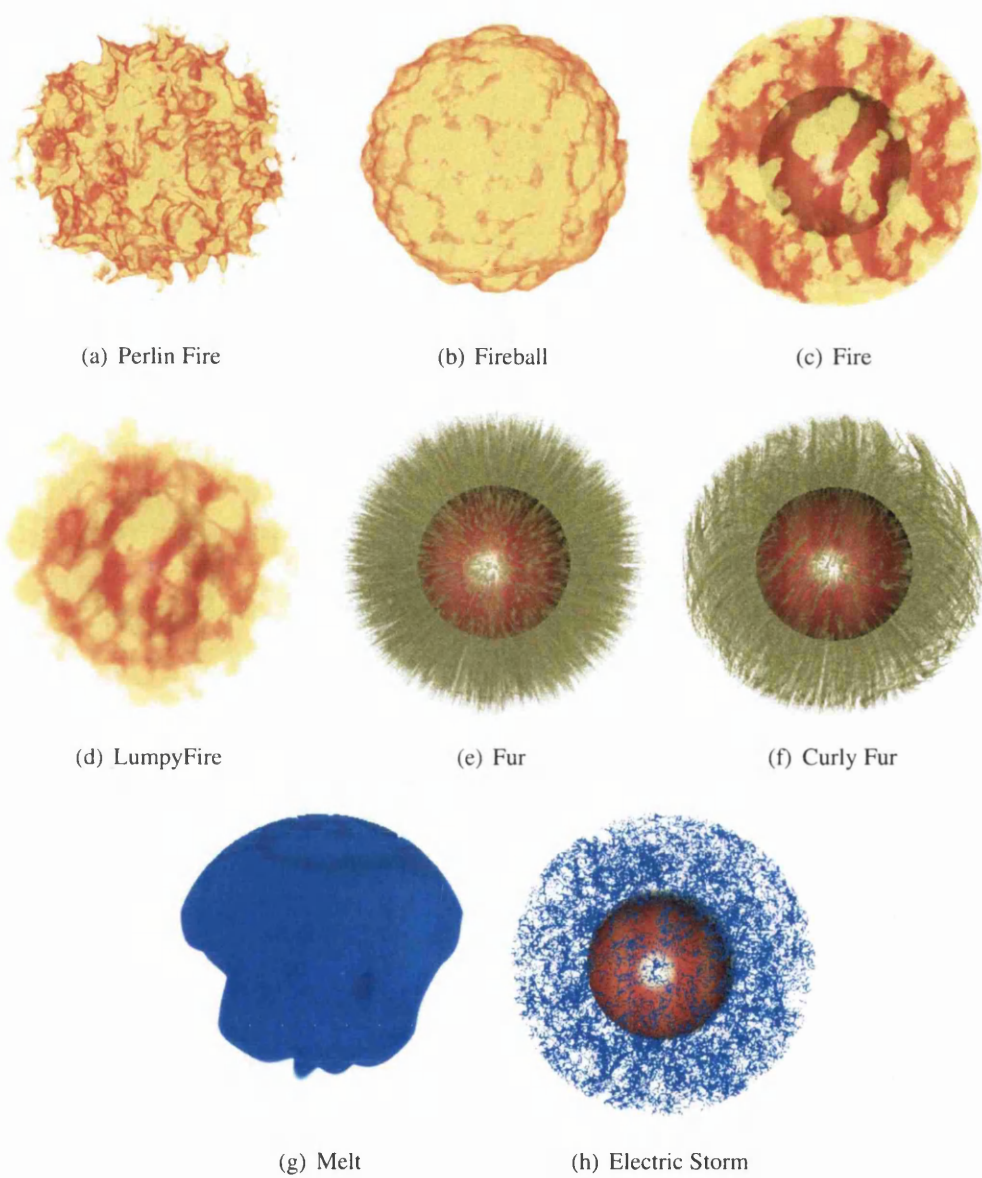


Figure 4.14: Examples of hypertexture applied to the *SphereDist* dataset

```

pixel fragmentShader(fragment, volume, transfer, density)
    voxel = volume(fragment.tex0)
    if (voxel.a >= density.z)
        pixel = transfer(hypertexture(fragment.tex0, voxel.xyz * 2.0 - 1.0,
            D(voxel.a), volume))
    else
        discard
    endif

```

Figure 4.15: OOP hypertexture fragment shader

```

pixel fragmentShader(fragment, volume, transfer, light, textureMatrix, density)
    voxel = volume(fragment.tex0)
    if (voxel.a >= density.a)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (voxel.a >= density.z)
        pixel = transfer(hypertexture(fragment.tex0, voxel.xyz * 2.0 - 1.0,
            D(voxel.a), volume))
    else
        discard
    endif

```

Figure 4.16: OOP hypertexture and iso-surface fragment shader

functions must be computed to correctly address the pre-integrated transfer function lookup table.

Since a segmentation of 3 distinct regions is performed before any classification of the soft-region, employing this technique can cause overlaps between the iso-surface and the soft-object. This is because the sample point is fetched along with the next sampling position along the ray. When including a lit iso-surface with the rendering this is avoided by using the slab iso-surface rendering method since both samples address a 2D texture map which allows a correct positioning of the iso-surface.

4.3.5 Object-Order Proxy Slice Hypertexture

OOP slice hypertexture effects are accomplished by directly evaluating the density modulation function in the fragment shader. Since the rendering strategy is mixed mode in respect of performing iso-surfacing for the original object definition and compositing for soft-region, the blending stage of the pipeline must be used to composite each sample into the frame buffer. The iso-surfacing of the original object definition is optional for hypertexture effects and both rendering styles are presented. Additionally slice and slab style rendering techniques are considered. Figure 4.3.5 is the iso-surface and hypertexture combined fragment shader, figure 4.3.5 is the soft-region only fragment shader. Figures 4.3.5 and 4.3.5 are the slab equivalent fragment shaders for iso-surfacing combined with hypertexture and hypertexture only respectively. The function $D(p)$ remaps the distance field scalar with the global *density* which contains iso-values for the iso-surface and soft-region boundaries.


```

pixel fragmentShader(fragment, volume, transfer, density)
    voxelf = volume(fragment.tex0)
    voxelb = volume(fragment.tex1)
    if (voxelf.a >= density.z)
        pixel = transfer(hypertexture(fragment.tex0, voxelf.xyz * 2.0 - 1.0,
            D(voxelb.a), volume), hypertexture(fragment.tex1,
            voxelb.xyz * 2.0 - 1.0, D(voxelb.a), volume))
    else
        discard
    endif

```

Figure 4.17: OOP *interploated* hypertexture fragment shader

```

pixel fragmentShader(fragment, volume, transfer, weight, light, textureMatrix,
    density)
    voxelf = volume(fragment.tex0)
    voxelb = volume(fragment.tex1)
    wght = weight(voxelf.a, voxelb.a)
    if (wght > 0.0)
        normal = (lerp(voxelf.xyz, voxelb.xyz, wght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (voxelf.a >= density.z)
        pixel = transfer(hypertexture(fragment.tex0, voxelf.xyz * 2.0 - 1.0,
            D(voxelb.a), volume), hypertexture(fragment.tex1,
            voxelb.xyz * 2.0 - 1.0, D(voxelb.a), volume))
    else
        discard
    endif

```

Figure 4.18: OOP *interploated* hypertexture and iso-surface fragment shader

4.3.6 Image-Order Single Pass Hypertexture

Two methods of hardware volume rendering allow empty space leaping. This can be accomplished using a min-max octree for standard volume datasets. Distance fields can also be used to accelerate ray-casting by skipping encountered distances along the ray. The original empty space skipping to an iso-surface is adjusted in respect of the soft-region boundary since a distance field will usually encode the original object with the value 0, inside voxels are given positive values and outside voxels are given negative values. Therefore an adjustment is required to correctly skip up to the soft-region. This can be achieved with the previously defined functions 3.3 or 3.4 by adjusting the iso-value to be the soft region boundary value.

The fragment shaders in figures 4.20 and 4.19 compute hypertexture with iso-surfacing and without respectively. Early ray termination is included in both shaders based upon the composited opacity value. Sampling is restricted to each sample point and no slab rendering is considered. Encountering the iso-surface allows an early ray termination due to the iso-surface being opaque. Figures 4.19 and 4.21 compute the slab alternative rendering strategies with an acceleration over OOP hypertexturing since volume lookups are reduced by reusing previous sampling positions along the ray. Previous hypertexturing results cannot be reused since the previous hypertexture result is not guaranteed to feature in the next soft-region sample point. Since a front and back distance value are required in slab rendering, no reuse of the previous volume lookups can be employed when computing empty space


```

pixel fragmentShader(fragment, volume, transfer, light, textureMatrix,
    density, dir)
    direction = dir(fragment.wpos)
    blend = (0.0, 0.0, 0.0, 0.0)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(raypos)
        if (voxel.a > density.z)
            output = transfer(hypertexture(raypos, voxel.xyz * 2.0 - 1.0,
                D(voxel.a), volume))
            blend = composite(output, blend)
        endif
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(raypos - fragment.tex0) || blend.a > 0.98)
            break
        endif
    endwhile
    pixel = blend

```

Figure 4.19: IOS hypertexture fragment shader

```

pixel fragmentShader(fragment, volume, transfer, light, textureMatrix,
    density, dir)
    direction = dir(fragment.wpos)
    blend = (0.0, 0.0, 0.0, 0.0)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(raypos)
        if (voxel.a > density.a)
            normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
            output = lighting(normal, fragment.tex1, light)
            blend = composite(output, blend)
            break
        else if (voxel.a > density.z)
            output = transfer(hypertexture(raypos, voxel.xyz * 2.0 - 1.0,
                D(voxel.a), volume))
            blend = composite(output, blend)
        endif
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(raypos - fragment.tex0) || blend.a > 0.98)
            break
        endif
    endwhile
    pixel = blend

```

Figure 4.20: IOS hypertexture and iso-surface fragment shader

```

pixel fragmentShader(fragment, volume, transfer, light, textureMatrix,
    density, dir)
    direction = dir(fragment.wpos)
    blend = (0.0, 0.0, 0.0, 0.0)
    rayPosf = fragment.tex0;
    rayPosb = rayPosf + direction;
    voxelb = volume(rayPosf)
    while (true)
        voxelb = volume(rayposb)
        if (voxelf.a > density.z)
            output = transfer(hypertexture(rayposf, voxelb.xyz * 2.0 - 1.0,
                D(voxelf.a), volume), hypertexture(rayposb,
                voxelb.xyz * 2.0 - 1.0, D(voxelb.a), volume))
            blend = composite(output, blend)
        endif
        rayPosf += direction;
        rayPosb += direction;
        voxelb = voxelb;
        .
        // further samples
        .
        if (direction.a < length(rayposf - fragment.tex0) || blend.a > 0.98)
            break
        endif
    endwhile
    pixel = blend

```

Figure 4.21: IOS *interpolated hypertexture fragment shader*

leaping. This extra burden of texture lookups in some cases outweighs the acceleration.

4.3.7 Results

Two rendering strategies are explored for complex hypertexturing volumetric objects, OOP rendering and IOS rendering. Post-classification and pre-integrated classification are explored for soft-region classification after segmentation using the object density function. Iso-surfacing is also considered for extended hypertexture effects. These methods are compared for performance and image quality characteristics. Gradient computation is not considered for the soft-region to perform lighting calculations, however iso-surfaces are subject to lighting. Frame rates are taken with no volume rotation to avoid memory access and cache issues.

Table 4.5 gives the frame rates obtained with the OOP rendering method and figures 4.23 and 4.24 displays the resultant images as applied to the *CTHeadDist* dataset and the *Sphere-Dist* dataset respectively. The complexity of the DMF applied to the soft-region has a drastic impact on rendering speed compared to standard iso-surface rendering or direct volume rendering. This is due to additional volume lookups being required and complex instruction combinations being required to compute arbitrary DMF functions. The conditional branching mechanism is utilised to bypass expensive hypertexture DMF functions, however throughput without conditional branching achieves similar frame rates for less complex hypertexture effects. This property allows hypertexture to be computed on older hardware that is not capable of dynamic branch instructions. In general hypertexture is accelerated by using dynamic conditional statements when there are multiple volume lookups and complex

```

pixel fragmentShader(fragment, volume, weight, transfer, light, textureMatrix,
    density, dir)
    direction = dir(fragment.wpos)
    blend = (0.0, 0.0, 0.0, 0.0)
    rayPosf = fragment.tex0;
    rayPosb = rayPosf + direction;
    voxelf = volume(rayPosf)
    while (true)
        voxelb = volume(rayposb)
        weight = weight(voxelf.a, voxelb.a).a
        if (weight > 0.0f)
            normal = (lerp(voxelf.xyz, voxelb.xyz, weight) * 2.0 - 1.0) *
                textureMatrix.inverseTranspose)
            output = lighting(normal, fragment.tex1, light)
            blend = composite(output, blend)
            break
        else if (voxelf.a > density.z)
            output = transfer(hypertexture(rayposf, voxelf.xyz * 2.0 - 1.0,
                D(voxelf.a), volume), hypertexture(rayposb,
                voxelb.xyz * 2.0 - 1.0, D(voxelb.a), volume))
            blend = composite(output, blend)
        endif
        rayPosf += direction;
        rayPosb += direction;
        voxelf = voxelb;
        .
        .    // further samples
        .
        if (direction.a < length(rayposf - fragment.tex0) || blend.a > 0.98)
            break
        endif
    endwhile
    pixel = blend

```

Figure 4.22: IOS interpolated hypertexture and iso-surface fragment shader

Dataset (size)	Viewport	Lit	Slices	Fireball		Melting		Fur	
				Iso	Int	Iso	Int	Iso	Int
<i>CTHeadDist</i> ($256^2 \times 128$) see Figure 4.24	512^2	No	128	48	27	27	17	42	23
			256	24	13	13	8	20	11
			512	12	6	6	4	10	5
			1024	6	3	3	2	5	2
		Yes	128	19	13	15	10	18	11
			256	9	6	7	5	8	5
			512	4	3	3	2	4	2
			1024	2	1	1	1	2	1
	1024^2	No	128	17	10	14	8	16	9
			256	9	5	7	4	8	4
			512	4	2	3	2	4	2
			1024	2	1	1	1	2	1
		Yes	128	7	5	6	4	6	4
			256	2	2	3	2	3	2
			512	1	1	1	1	1	1
			1024	< 1	< 1	< 1	< 1	< 1	< 1
<i>SphereDist</i> (256^3) see Figure 4.23	512^2	No	128	52	29	24	14	45	23
			256	25	13	12	7	22	12
			512	12	7	6	3	10	6
			1024	6	3	3	1	5	3
		Yes	128	22	14	17	12	20	14
			256	12	6	6	5	10	6
			512	5	3	4	2	5	3
			1024	2	1	2	1	2	1
	1024^2	No	128	18	11	13	8	15	9
			256	9	5	6	4	7	4
			512	4	2	3	2	3	2
			1024	2	1	1	1	1	1
		Yes	128	8	5	7	4	7	5
			256	4	2	3	2	3	2
			512	2	1	1	1	1	1
			1024	1	< 1	< 1	< 1	< 1	< 1

Table 4.5: OOP hypertexture frame rates in frames per second, Iso is single sample hypertexturing with post-classification in the soft-region and Int is interpolated hypertexture rendering with pre-integrated classification in the soft-region. Non lit variants do not contribute an iso-surface whilst lit variants contribute an iso-surface or interpolated iso-surface. All rates are rounded down.

instruction combinations.

Table 4.6 gives the frame rates for IOS hypertexturing. These frame rates do not include empty space leaping. Table 4.7 outlines the rendering speeds possible with space leaping for the IOS approach. Frame rates compared to the standard IOS approach for fireball effects are comparable, however degrade with more samples in a less linear nature. Dynamic branching costs currently make the performance of the slab rendering strategy with empty space leaping slower since they are required to skip expensive computations of two hypertexture functions. This method is still considered to be the best approach since future hardware will improve on branching performance. Additionally the early ray termination combined with empty space leaping for iso-surfacing during hypertexture rendering outperforms the non iso-surface consideration alternative due to early ray termination being performed when the iso-surface is intersected. This is due to the early ray termination upon reaching the iso-surface and not considering the soft-region positioned beyond the iso-surface encountered. The heavy cost of additional texture access and dynamic branches in the slab style of rendering generates many more instructions which degrade performance drastically.

The difference between slice and slab rendering techniques are apparent for sparse soft-regions of high frequencies that are generally under-sampled with post-classification. Fur, curly fur and electric storm hypertextures are generally under-sampled with post-classification unless a small step size is used. Pre-classification offers a performance increase in these cases alone with comparable image quality. Many effects including fire effects do not benefit from the pre-integrated classification step and the additional burden of computing two DMF functions per sample. This is due to the highly stochastic properties of these effects. Figure 4.24 contains images of the fireball effect with post-classification and pre-integrated classification. There is only a small variation between output images where under-sampling and aliasing artefacts are consistent at any sampling level, each sampling level results in aesthetically pleasing results. A clear benefit to pre-integrated classification can be seen in figure 4.23 for fur hypertexture. The transfer function contained high frequencies which post-classification under-samples with few samples through the soft-region. However changing the transfer function to contain no high frequencies allows the post-classification method to obtain the desired results (see Figure 4.23(i)).

OOP hypertexture rendering currently outperforms image-order approaches that include empty space leaping and early ray termination due to less cycles being used during fragment shading. OOP rendering do not compute true 32 bit blending where IOS techniques enable blending at full precision. This is especially important for high-frequencies throughout the soft-region. Therefore rendered images are subject to artifacts due to quantised blending. It is further evident that IOS hypertexturing will outperform OOP methods as conditional branching hardware matures because the linear nature of the frame rates is reduced by utilising the image-order acceleration techniques.

4.4 Animation Techniques

Animation methods for the outlined procedural texturing methods are based on varying parameters to the functions over time. This enables automatic animation of the model and

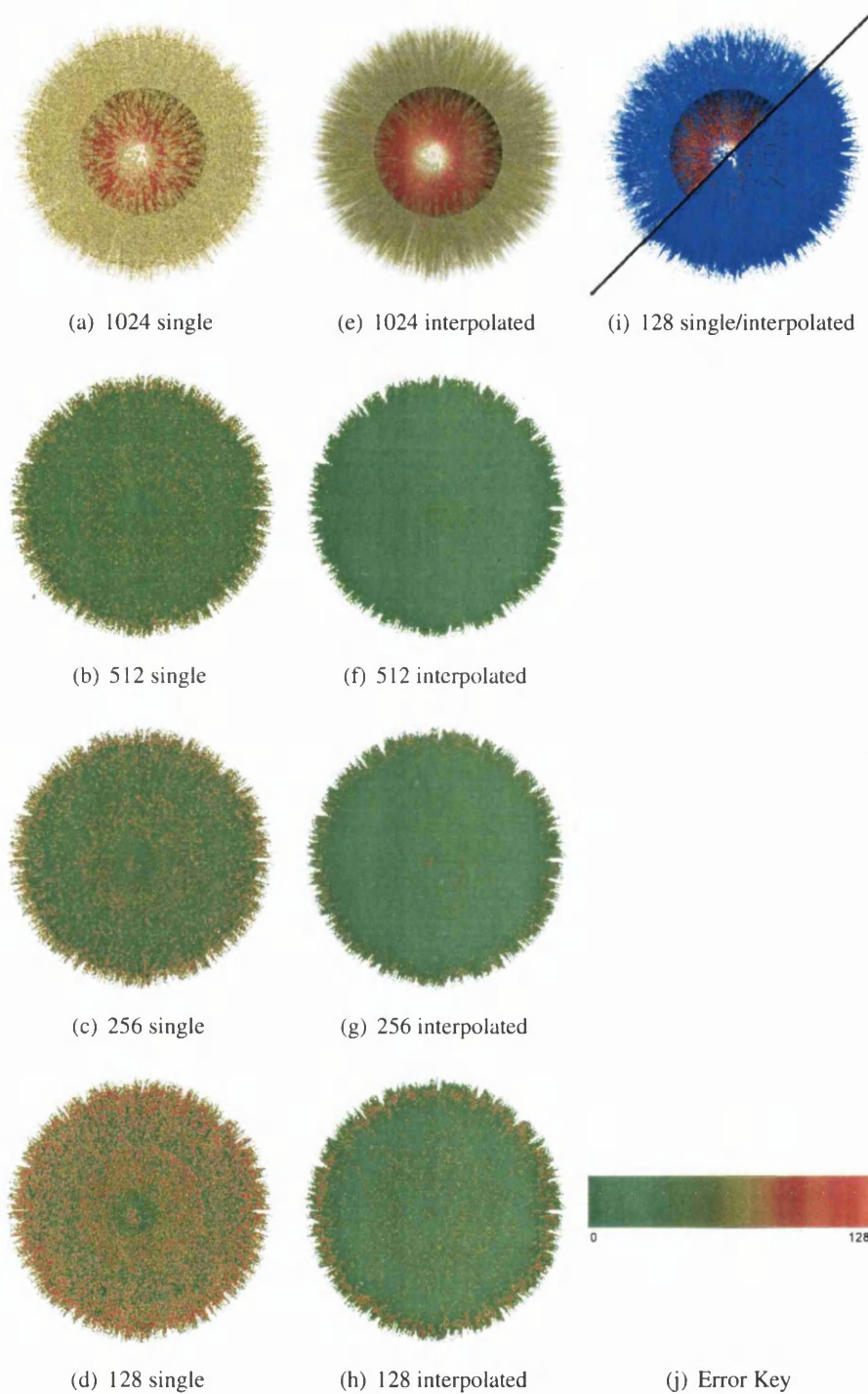


Figure 4.23: *SphereDist* dataset fur hypertexture (a) to (d) and interpolated hypertexture with iso-surface(e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared highlight sampling differences. and the error range for difference images is given in (j)

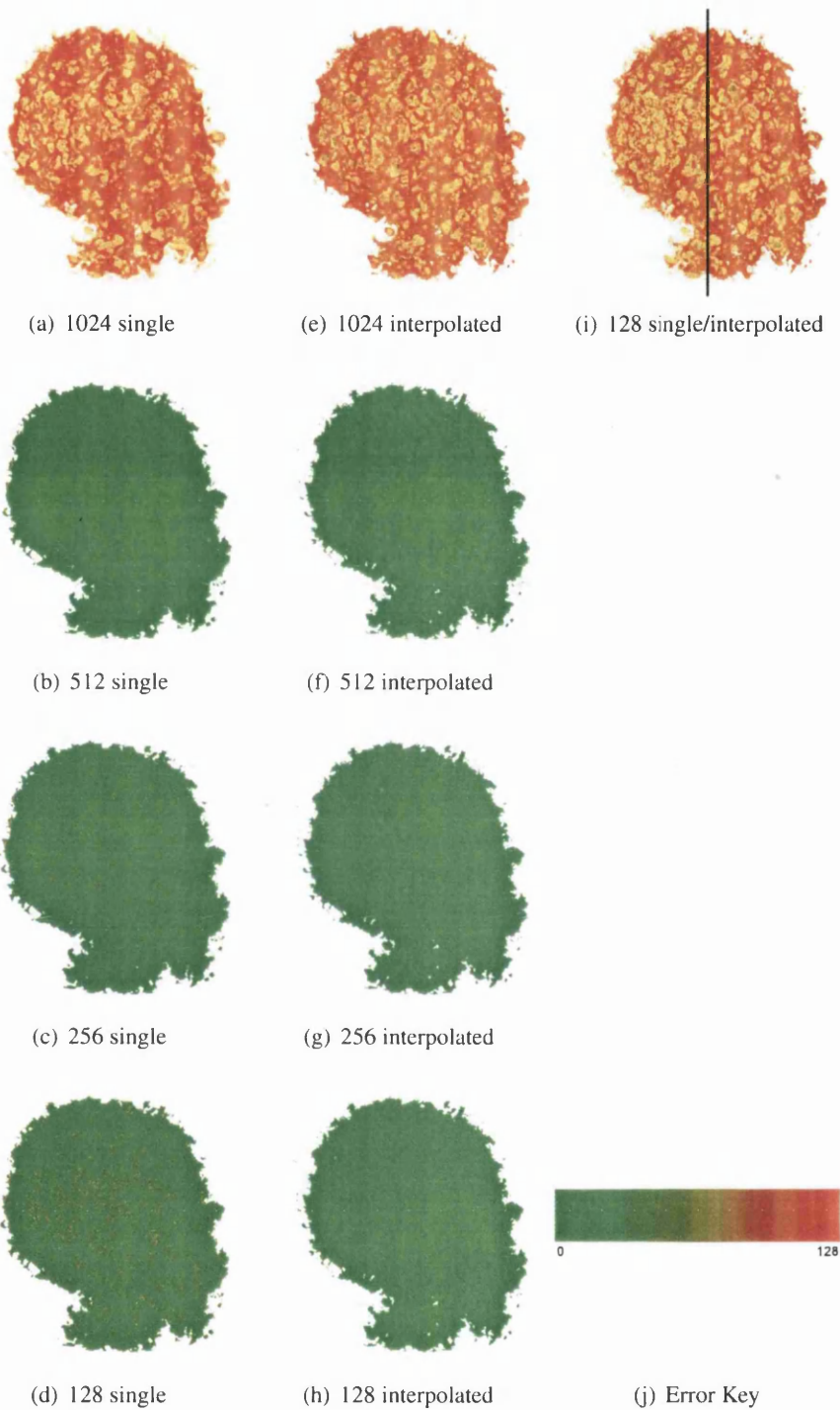


Figure 4.24: *CTHeadDist* dataset fireball hypertexture (a) to (d) and interpolated hypertexture (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

Dataset (size)	Viewport	Lit	Slices	Fireball		Melting		Fur	
				Iso	Int	Iso	Int	Iso	Int
<i>CTheadDist</i> ($256^2 \times 128$) see Figure 4.24	512^2	No	128	29	22	23	20	27	20
			256	15	11	11	9	13	10
			512	7	6	5	4	6	5
			1024	3	3	2	2	3	2
		Yes	128	17	13	12	9	12	10
			256	8	6	6	4	6	5
			512	4	3	3	2	3	2
			1024	2	1	1	1	1	1
	1024^2	No	128	12	6	8	5	10	6
			256	6	3	4	2	5	3
			512	3	1	2	1	2	1
			1024	1	< 1	1	< 1	1	< 1
		Yes	128	9	5	6	4	8	4
			256	4	2	3	2	4	2
			512	2	1	2	1	2	1
			1024	1	< 1	1	< 1	1	< 1
<i>SphereDist</i> (256^3) see Figure 4.23	512^2	No	128	27	21	21	17	25	19
			256	14	12	10	7	12	10
			512	7	6	5	3	6	5
			1024	3	3	2	1	3	2
		Yes	128	14	12	11	8	12	10
			256	7	6	5	4	6	5
			512	3	3	2	2	3	2
			1024	1	1	1	1	1	1
	1024^2	No	128	8	5	7	5	8	5
			256	3	2	3	2	3	2
			512	1	1	1	1	1	1
			1024	< 1	< 1	< 1	< 1	< 1	< 1
		Yes	128	6	5	5	4	6	5
			256	3	2	2	2	3	2
			512	1	1	1	1	1	1
			1024	< 1	< 1	< 1	< 1	< 1	< 1

Table 4.6: IOS hypertexture frame rates in frames per second, *Iso* is single sample hypertexturing with post-classification in the soft-region and *Int* is interpolated hypertexturing with pre-integrated classification in the soft-region. Non lit variants do not contribute an iso-surface whilst lit variants contribute an iso-surface or interpolated iso-surface. All rates are rounded down.

Dataset (size)	Viewport	Slices	Non Lit		Lit	
			Iso	Int	Iso	Int
<i>CTHeadDist</i> ($256^2 \times 128$) see Figure 4.24	512^2	128	28	24	28	16
		256	18	15	24	8
		512	12	9	16	5
		1024	8	6	11	3
	1024^2	128	15	10	14	6
		256	8	7	10	3
		512	5	4	7	2
		1024	3	2	5	1

Table 4.7: IOS hypertexture frame rates with empty space leaping for the fireball hypertexture in frames per second, Iso is single sample hypertexturing with post-classification in the soft-region and Int is interpolated hypertexturing with pre-integrated classification in the soft-region. Non lit variants do not contribute an iso-surface whilst lit variants contribute an iso-surface or interpolated iso-surface. All rates are rounded down.

effect being rendered and removes burden to texture artists and animators. The goal is to achieve a realistic looking sequence of images without artifacts in real-time.

Animation of procedural texturing techniques can be performed in a variety of ways. The methods described here are all parameter based and exhibit differing complexity. Altering the texture domain is detailed in section 4.4.1, providing higher order noise implementations is presented in section 4.4.2 and additional hypertexture effects are described in section 4.4.3.

The procedural texturing techniques presented are all view-independent which allows arbitrary viewing parameters and texture space manipulations to be defined whilst still maintaining high-quality rendering without artifacts. This gives massive scope to procedurally generated animations without texture artists having to manipulate separate key frames.

4.4.1 Texture Domain

The most simple and runtime efficient method of animating procedural textures is to provide a mechanism for offsetting or transforming the noise domain. The descriptions of solid texturing and hypertexturing include the ability to transform positions used in noise contributions. Simply providing a mechanism to allow arbitrary matrix multiplication allows a vast array of differing offset strategies to be employed such as translation, scaling and rotation (see Eqn 4.15).

$$T(x) = A x \quad (4.15)$$

where A is the concatenated transformation matrix containing translation, rotation and scale contributions and x is the co-ordinate to transform.

OOP techniques benefit from providing proxy slice geometry which can be rasterized with two sets of texture coordinates. The first set is used to address the volume being sampled

whilst the additional set of texture coordinates is used to address texture space. IOS methods do not benefit from the ability to provide two sets of texture coordinates without performing an additional rendering pass or providing a matrix to the fragment shader to multiply with each position encountered for addressing the texture domain. The concatenation of matrices can be performed on the CPU and uploaded to the GPU when changes occur. Usage of the CPU in this case reduces the GPU overhead by reducing the amount of instructions executed for each sample position. The provision of the texturing co-ordinates effectively maintains the same instruction count with the addition of a texture lookup into texture space. Computing this position on the fly incurs a matrix multiply which will require 4 extra instructions.

This technique of offsetting the texturing co-ordinate set with matrix algebra requires that the transformed sample points are contained within the textures original domain. Procedural generation of texture space guarantees this property since it will exhibit in infinite domain, however a texture map based implementation has a finite domain.

A simple alternative to providing a complete matrix to the GPU for arbitrary transformation instead provides a small offset to the texture co-ordinates. Generally one direction is chosen and the offset scalar uploaded to the GPU as a global variable. This enables performance to be maintained at the same level as not requiring animation.

Generally these strategies to providing animation can be computed with minimal overhead and provide a simplistic animation. The finite texture domain of the fast procedural implementations introduces the problem of addressing outside the texture space which requires a periodic function such as *sin* to control the offset. Mirrored repeat textures can also be used to allow a continued infinite domain. These techniques impact the overall visual non-repeating pattern of the noise primitive.

4.4.2 Higher-Order Noise Primitives

Procedural texture synthesis that is based on the noise function can additionally be animated by adding a further dimension to the noise domain to represent time. Procedural noise techniques incur an additional performance penalty since a 4D lattice must be considered in replacement of the 3D counterpart. The noise algorithm's complexity grows from interpolating 8 vertices to interpolating 16 vertices which adds an order of magnitude to the algorithm. Techniques such as solid texturing can be rendered in real-time when using procedural noise implementations since only one noise lookup is performed per ray when using deferred shading techniques such as the IOS approach. This approach also benefits from empty space leaping strategies.

GPU hardware does not allow native 4D texture maps to be uploaded and thus must be a modified 3D texture. Computing several 3D blocks of noise with a 4D noise implementation allows a finite 4D texture domain when one sampling direction is the concatenation of each noise block. This proves to be fast to render since the texture coordinates can be rescaled in one direction in respect of the total number of noise blocks contained in the texture. This rescaling is then multiplied cheaply with a global representing time to perform the correct addressing in texture space (see Eqn 4.16). Since a finite 4D texture block is provided, using a mirrored repeating texture strategy allows the general visually non-repeating

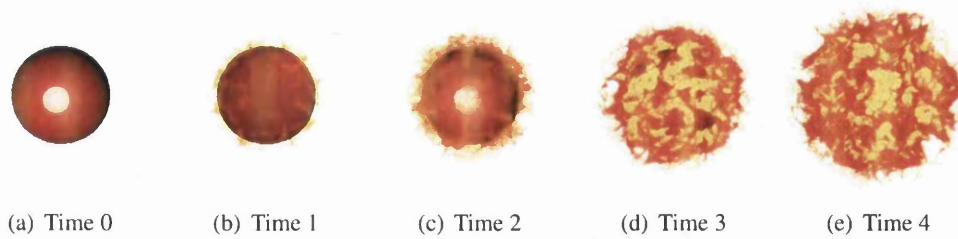


Figure 4.25: Animation of PerlinFire at differing time intervals

pattern property to be maintained in the direction providing the 4th dimension.

$$p' = \frac{p_x}{n} + \frac{1}{t}n \quad (4.16)$$

where p' is the new texturing coordinate, p is the original texture co-ordinate in the $[0, 1]^3$ range, $t \in \mathbb{N}$ is the number of 3D texture blocks building up the 4D noise function and $n \in \mathbb{N}$ is the current texture block for consideration or animation frame. The $\frac{1}{t}n$ can be computed on the CPU to reduce fragment shader instructions. Additionally when using a mirrored repeat mode, $n > t$ is valid.

4.4.3 Hypertexture Parameters

Hypertexture effects can also benefit from controlled manipulation of the remaining rendering properties such as bias, gain and the object density function over time. Constantly differing animations can be achieved using the methods described above with the added ability to define complex animations such as explosions by varying parameters to the hypertexture DMF function at precise intervals. This analogous to key-frame animation is easy to implement, intuitive and does not require extensive manipulation at key frames.

A simple explosion effect may be generated by altering the soft-region iso-value to expand over time using the fireball hypertexture. More complex extensions may also be computed by altering bias and gain over time to alter the way the explosion occurs through the expanding soft-region. In combination with a continuously varying 4D noise function a natural looking complex animation can be created by varying only 4 parameters. These manipulations can occur at different times to control the exact progress of the animation and provide an animator fine control.

Real-time rendering can be maintained with these precise animations by utilizing 4D texture noise primitives and repeating texture access modes. Figure 4.25 provides frames from an explosion hypertexture with varying soft-region iso value, bias, gain and 4D noise.

4.5 Summary

This chapter has introduced flexible and scalable real-time procedural texture synthesis for volume graphics. Solid texture and hypertexture have been explored as powerful techniques to add rich detail to a volume object. The focus of this chapter has been to provide procedural texturing effects capable of surface and object definitions in real-time to the volume graphics domain using GPU hardware. Previous work in this field has not enabled real-time rendering and as a result this description of procedural texturing for volumes has a wide reaching level of applications in the visualisation, modelling, animation and entertainment industries.

The emergence of real-time volume techniques make it possible to define a real-time volumetric pipeline to provide a richer architecture for computing general graphics imagery. Additionally the strength of the volume graphics approach is evident in the modelling of amorphous phenomena and constant complexity for increasingly complex objects. Additionally speed-up mechanisms have been shown to be possible to implement on GPU hardware to further accelerate these techniques.

Particular attention has been focused to enabling procedural modelling using primitive functions such as noise and turbulence in the fastest and most aesthetically pleasing manner available, efficient rendering of these effects and a generalized pipeline to include arbitrary functions at run-time. These techniques can be used in software that provides volume graphics modelling environments among other applications of volumetric effects.

Parts of this work have been presented at the 4th *International Workshop on Volume Graphics* and also published in *Volume Graphics 2005*.

Chapter 5

Volume Surface Detail

Contents

5.1	2D Texture Mapping	153
5.2	Tangent Space	163
5.3	Bump Mapping	168
5.4	Displacement Mapping	183
5.5	Summary	196

This chapter explores fine surface detail rendering techniques for volume datasets in real-time. The goal is to define additional tools to compute surface detail to enable a rich volume graphics framework for general 3D surface synthesis. Visual interpretation of surface texture is important to synthesize natural looking images. The emergence of volume approaches as a graphics primitive benefits from the ability to model complex surface structures efficiently and intuitively whilst additionally providing analogous cases to surface graphics techniques. Surface detail may be added to the volume graphics pipeline with the addition of 2D texture mapping techniques. This chapter introduces GPU accelerated volume texture mapping with the addition of more advanced surface detail techniques.

Many real world objects exhibit differing colours and attributes over their surfaces. For example a human has very subtle skin tone variations and small lines, wrinkles and pores in the skin, as well as small hairs and inconsistent densities across the surface of the body. A realistic rendering of such an object must convey these small irregularities from a perfectly curved surface to the user for intuitive recognition of the subject. The computer graphics model can treat surfaces as smooth uniform entities that do not contain discontinuity. Procedural texturing techniques do not provide a means of intuitively representing all surface texture as often mathematical models cannot account for such detail. This short fall in procedural synthesis is explored in this chapter. It is therefore important to provide techniques to allow computer graphics algorithms to define these properties and surface texturing techniques [Cat74, Cat75, Bli78a] are defined and expanded for this purpose.

Providing a mechanism to keep object shape and final surface detail separate allows models to be reused and resemble different materials and properties by changing the texturing

environment alone. In volume graphics, this can be an important feature of a modelling environment where constant re-voxelisation of an object into different forms is not practical. For example it might be desirable to represent a sphere object as solid marble in one modelling instance which can utilise solid texturing to convey a solid marble sphere, it might then be necessary to represent a sphere as a spiky object in another modelling instance to convey a weapon in a games environment, a sphere with different densities in its construction that is weathered in an outdoor scene or simply changing the underlying simple shape by manipulating the modelling parameters. The underlying sphere does not change in these instances, but can be textured or manipulated to model differing properties depending on the application. Current techniques such as CVG require several datasets being used for this purpose or the alternative is to re-voxelise a dataset which requires the detail to be mapped to the surface in a specific way. These approaches therefore keep object detail separate from texture detail which is an important notion in intuitively representing arbitrary objects that have similar underlying shape detail.

Volume datasets exhibit less key attributes to enable 2D texture mapping and this is firstly explored in 5.1. Section 5.2 provides an algorithm to compute a tangent space to object space mapping for vectors, a technique used in the following sections for defining lighting techniques. Sections 5.3 to 5.3.3 explore *Bump mapping*, a technique to add small irregular detail to surfaces that removes the uniform appearance of smooth surfaces with lighting. Sections 5.4.1 to 5.4.3 explore *Displacement mapping*, an extension of bump mapping that removes some restrictions imposed by this technique and finally a summary is presented in section 5.5.

5.1 2D Texture Mapping

This section explores the mechanism to define a uv parameterization of a volume's iso-surface and introduces the first accelerated GPU implementation of these techniques for volume objects. In general a function $t : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ is required that maps each point on an objects surface to a unique position in 2D texture space. Therefore texture patches cannot be utilized since there is no flat geometric primitives to project into texture space.

The Bier and Sloan [BS86] two part texture mapping algorithm is employed for this purpose since a volume dataset does not contain a geometric representation. An intermediate parametrically defined object is used to form this as a geometric representation is present. Volume datasets can be projectively textured in the same manner as surface based geometry that exhibits no uv parametrisation.

Winter [Win02] introduced a volume graphics implementation of two part texture mapping in software. The two part texture mapping algorithm is implemented as a set of functions for intermediate surfaces that are applied during iso-surface classification. Winter and Chen [WC01] describe the *vlib* API which includes a software implementation of 2D texturing using this technique.

Shen and Willis [SW05] also describe the use of two part texture mapping for volume objects. Extensions to the two part texture mapping algorithm are explored to perform anti-

aliasing and increase the level of accuracy during magnification mappings.

The two part texture mapping techniques of a forward mapping from texture space to object space and additionally an inverse or backward mapping from object space to texture space are explored and defined. Section 5.1.1 defines forward mapping enabling computation of 2D templates to aid texture artists to morph and paint on known positions of the volume's iso-surface. Section 5.1.2 defines backward mapping which allows the application of a texture map to a volume object. Section 5.1.3 defines the parametrically defined intermediate surface geometries used in this approach. Section 5.1.4 and 5.1.5 introduce the first GPU 2D texture mapping implementations using these techniques. Finally section 5.1.6 provides performance measurements for each GPU approach.

5.1.1 Forward Mapping

Forward mapping is the process of mapping from unit square texture space to points on the object's surface (object space). For volume rendering, rays must be cast into the volume in some manner to intersect the iso-surface for consideration. This forward mapping therefore employs ray-casting from texture space locations through the object space to intersect the iso-surface.

Forward Mapping is defined by firstly mapping from texture space locations to intermediate object surface points (see Function 5.1)

$$S : \mathbb{R}^2 \rightarrow \mathbb{R}^3 \quad (5.1)$$

A Mapping from the parametric intermediate surface is then made to the object being rendered (see Function 5.2).

$$O : \mathbb{R}^3 \rightarrow \mathbb{R}^3 \quad (5.2)$$

A forward mapping is thus the combination of the S and O functions (see Eqn 5.3).

$$O(S(u, v)) = p \quad (5.3)$$

where $p \in \mathbb{R}^3$ represents a point on the objects surface and u, v represent the position in texture space for a given texel.

The S mapping can be defined parametrically and therefore a suitable method for computing O must be used. There are four differing approaches to compute O (see Figure 5.1):

- Reflected Ray - A ray that intersects the intermediate object's surface is reflected around the normal vector at the intersection point. The intersection point on the object's surface is defined from the reflected ray.
- Intermediate Surface Normal - A point on the objects surface that is intersected by a normal from the intermediate surface.
- Object Normal - The intersection of the object's surface normal vector with the intermediate surface.

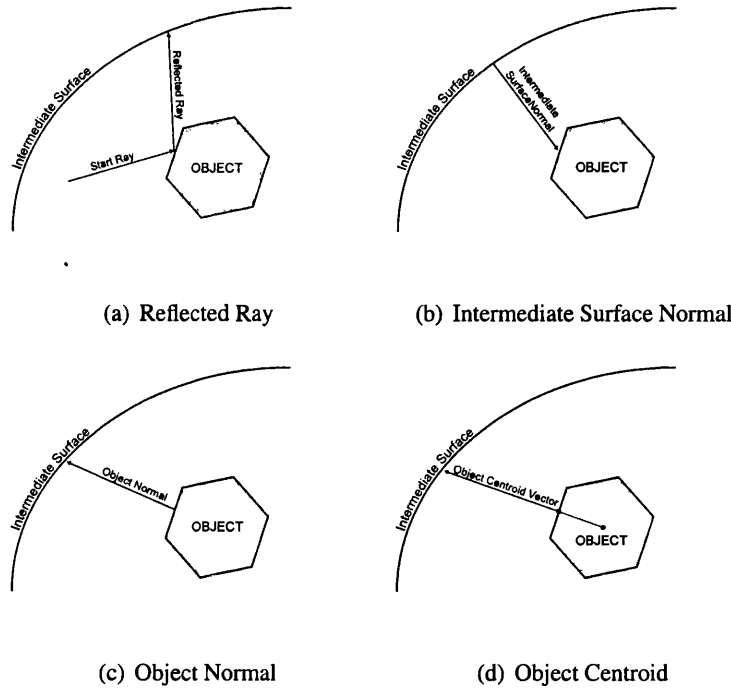
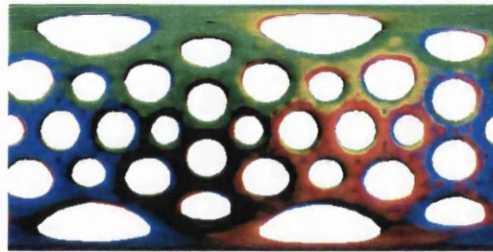


Figure 5.1: Mapping strategies for O and O^{-1}

- Object Centroid - project from the centre of the object or objects coordinate space through the point on the surface being considered onto the intermediate parametric surface.

The reflected ray method is not valid for template rendering since the object space starting ray is unknown and no one-to-one or many-to-one relationships can be guaranteed for arbitrary rays cast from the intermediate surface to the object's surface. The intermediate surface normal method is well suited to the problem of casting rays into the volume dataset since disjoint rays can be cast from the intermediate surface into the volume. A many-to-one or one-to-one relationship can be maintained. Both the object normal and object centroid methods are also not well defined since there may be several points on the object that project onto one intermediate surface point which will result in ambiguities. Additionally rendering the template is complicated further by firstly having to traverse the volume in a standard method and then perform an additional ray-casting towards the intermediate surface when the iso-surface is encountered.

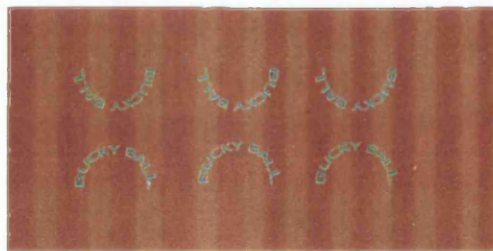
By using the intermediate surface normal a guaranteed many-to-one relationship between points on the intermediate surface and points on the object's surface is guaranteed. Additionally volume rendering can be performed using the intermediate surface normals as ray directions to intersect the iso-surface. This approach benefits from the ability to cast rays from template image final pixels by first performing the S mapping and ray-casting along the normal vector encountered on the intermediate surface. Figure 5.2(a) demonstrates a sphere mapping approach and the resulting template extracted from the *BuckyBall* dataset. The surface normals encountered at the iso-surface are used to provide detailed information



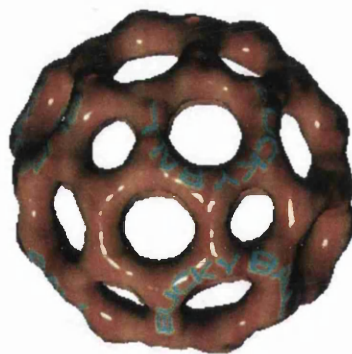
(a) Template



(b) Template Painting



(c) Final Texture Map



(d) Rendered Result

Figure 5.2: Painting on the surface of the *BuckyBall* dataset. The template is created with ray casting into the volume dataset with forward mapping (a), a texture artist then paints on the template using layers to separate the texture map from the template (b), the texture layer is stored as a texture map (c) and the texture map is applied to the original volume object and rendered (d).

to the texture artist.

5.1.2 Backward Mapping

Generally in ray-casting the opposite problem to template rendering is to be evaluated since during ray traversal, points are encountered on the objects iso-surface and require a uv parameterization in order to address a 2D texture map. The inverse or backward mapping is the combination of inverse functions $S^{-1} : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ and $O^{-1} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ (see Eqn 5.4)

$$S^{-1}(O^{-1}(x, y, z)) = t \quad (5.4)$$

where $t \in \mathbb{R}^2$ represents the texel location or u and v parameters.

The original algorithm contains four separate definitions for O^{-1} . This mapping from the original object to the intermediate parametric surface in object space can be defined with:

There are four differing approaches to compute O^{-1} (see Figure 5.1).

- Reflected Ray - A ray that intersects the intermediate object's surface is reflected around the normal vector at the intersection point. The intersection point on the object's surface is defined from the reflected ray.
- Intermediate Surface Normal - A point on the objects surface that is intersected by a normal from the intermediate surface.
- Object Normal - The intersection of the objects surface normal vector with the intermediate surface.
- Object Centroid - project from the centre of the object or objects coordinate space through the point on the surface being considered onto the intermediate parametric surface.

Generally during ray-casting the object normal, object centroid or intermediate surface normal methods can be used. This is due to the object gradient normal being known at iso-surface intersection, the object's centre point being known or estimated and a parametric mapping onto a non-unit sphere sharing it's intermediate surface normal. The object centroid can be approximated, or defined in some cases by analysing the volume dataset. For example the maximal distance value in a distance field should represent the centre point of an object. The object normal method exhibits a many-to-one correspondence since normals from two differing samples can map to the same point on the intermediate surface. The object centroid method provides an analogous to the intermediate surface normal method since a centroid located at the intermediate surface normal centre will produce the same vectors.

With intermediate surface normals there is a potential for multiple intermediate surface normal vectors to intersect the object's surface position. The reflected ray method exhibits the problem of firstly defining the starting ray which is unknown at sample locations. If template images are created by intermediate surface normals either the object centroid or intermediate surface normal mappings can be used. It is also more efficient to use these methods

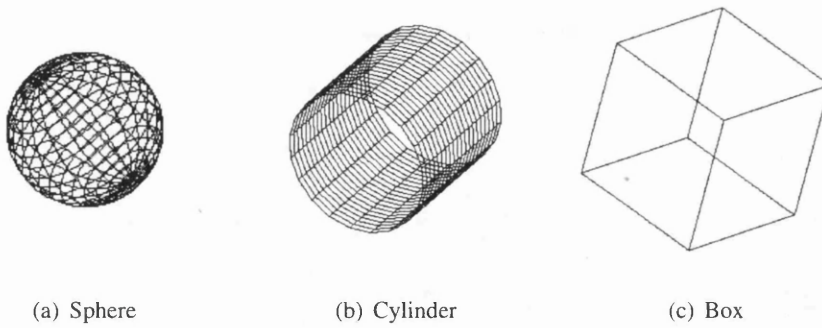


Figure 5.3: *Intermediate surface geometries*

since the location on the surface in object space and the gradient normal are required to map to the intermediate surface with the object normal method. In practice ray-casting is required to intersect the intermediate surface.

An example template, template painting, resultant texture and final image are given in figure 5.2

5.1.3 Intermediate Parametric Surfaces

Intermediate surfaces should closely approximate the underlying object being texture mapped and additionally be parametrically defined. Therefore no consideration is made for planes since volume datasets are 3D in nature. Additionally the box or cube intermediate surfaces are not considered since an explicit mapping function must be defined for cube vertices. The conditionals required by the mapping functions make it inefficient to compute at interactive rates on GPU hardware. The intermediate surface normal representation is used for both mapping functions since this provides a coherency and additionally a sphere of arbitrary radius contains the same normal vectors, allowing direct computation of the mapping function without projecting onto the object's surface along the intermediate surface's normal. The two main intermediate geometries are considered for this purpose are (see Figure 5.3):

- Sphere
- Cylinder

The descriptions below assume that texture co-ordinate space is defined in the unit square $u, v \in [0, 1]^2$ and the object space is defined in the unit cube $x, y, z \in [0, 1]^3$.

A sphere of centre (x_0, y_0, z_0) and radius r is defined as a set of points $s \in \mathbb{R}^3$ that satisfy equation 5.5.

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2 \quad (5.5)$$

Parametrically a point is defined on the sphere at the origin(see Eqn 5.6):

$$(r \cos(\theta) \sin(\phi), r \sin(\theta) \sin(\phi), r \cos(\phi)) \quad (5.6)$$

where r is the radius, $\theta \in [0, 2\pi]$ is an azimuthal (longitude) co-ordinate and $\phi \in [0, \pi]$ is a polar (colatitude) co-ordinate.

The mapping function S for a unit sphere with centre at $(0.5, 0.5, 0.5)$ is thus defined to bound the unit cube (see Eqn 5.7):

$$S(u, v) = (0.5 + \cos(\theta) \sin(\phi), 0.5 + \sin(\theta) \sin(\phi), 0.5 + \cos(\phi)) \quad (5.7)$$

where $\theta = 2\pi u$ and $\phi = \pi v$.

When rendering a template, the unit square is rasterized and application of this function defines the starting co-ordinates in object space (or the intermediate surface point) of each ray. The ray direction is defined by observing that any normal vector from the intermediate surface will point to the object space origin $(0.5, 0.5, 0.5)$. Thus the ray direction is $(0.5, 0.5, 0.5) - S(u, v)$.

The reverse mapping is performed with respect to the co-ordinate system defining the sphere centre at $(0.5, 0.5, 0.5)$ (see Eqn 5.8):

$$S^{-1}(u, v) = \left(\frac{\arctan\left(\frac{x-0.5}{z-0.5}\right)}{2\pi}, \frac{\arccos\left(\frac{y-0.5}{\sqrt{(x-0.5)^2 + (y-0.5)^2 + (z-0.5)^2}}\right)}{\pi} \right) \quad (5.8)$$

A O^{-1} mapping is not required since it can be observed that a normal vector on a sphere with an arbitrary radius will be the same. Therefore any point along this vector will produce the same result.

A cylinder of radius r and height h with height being aligned with the z axis is parametrically defined as a set of points $c \in \mathbb{R}^3$ that satisfy the equation 5.9.

$$(r \cos(\theta), r \sin(\theta), z) \quad (5.9)$$

where $\theta \in [0, 2\pi]$ and $z \in [0, h]$.

The orientation of the cylinder must be described when mapping the object. A vertical cylinder is used here for illustration where its base is defined at $(0.5, 0, 0.5)$. The cylinder's radius must bound the unit cube. S is then defined (see Eqn 5.10)

$$S(u, v) = (0.5 - \sin(\theta), h, 0.5 + \cos(\theta)) \quad (5.10)$$

where $\theta = 2\pi u$ and $h = 1.0 - z$.

When rendering a template, the unit square is rasterized and application of this function defines the starting co-ordinates in object space (or the intermediate surface point) of each

ray. The ray direction is defined by observing that any normal vector from the intermediate surface will point to the object in the same manner. The single point from sphere mapping is replaced with a point on a line defined at $(0.5, y, 0.5)$. Thus each ray direction can be calculated with $(0.5, y, 0.5) - S(u, v)$ where y is the same value returned from $S(u, v)$.

The reverse mapping is performed with respect to the co-ordinate system defining the centre of the cylinder at $(0.5, 0.5, 0.5)$ (see Eqn 5.11).

$$S(x, y, z) = \left(\frac{\arctan\left(\frac{x-0.5}{z-0.5}\right)}{2\pi}, 1.0 - y \right) \quad (5.11)$$

A O^{-1} mapping is again not required since it can be observed that a normal vector on a cylinder with an arbitrary radius will be the same. Additionally the height of the cylinder will not affect this property.

5.1.4 Object-Order Proxy Slice 2D Texturing

Two methods are included for applying texture maps to volumetric objects with intermediate surfaces. The slice and slab methods can both be used since 2D texturing is an extension of iso-surface rendering. A function *computeUV*(p) is supplied to the fragment shader to allow arbitrary uses of a reverse mapping function. This function can be changed depending on the proxy geometry that is required for rendering. Additionally the 2D texture map is provided as the function *textureMap*(p). A speed-up mechanism is possible by providing a pre-computed lookup table defining the unit cube with each position containing a u and v parameter in separate channels. Alternating between intermediate surface geometries is possible by either switching the pre-computed texture map or changing the function's definition and re-compiling the fragment shader at runtime. A large volume dataset is required to be uploaded to the GPU in order to achieve a speed-up with comparable image quality. Testing has shown that a reasonably sized lookup table introduces errors due to the quantising of float values from 32 bits to either 8 or 16 bits. It is expected that future hardware will allow true 32 bit texturing where this speed up method can be used without loss of image quality. These speed-ups are therefore not included in future discussions.

The vertex shader for use in 2D texturing is the same as the standard iso-surfacing vertex shader given in 3.20. No blending is computed during this iso-surfacing algorithm although blending must be introduced if a semi-transparent iso-surface is required. Figure 5.4 is the slice based iso-surfacing technique of 3.25 that is adapted to include 2D texturing. The slab extension is depicted in figure 5.5 which is an adaptation of the slab iso-surfacing technique of 3.27. There is no requirement for the colour table in this case since only the interpolation factors are required to compute the normal and position exactly on the iso-surface. Therefore this method benefits from increased accuracy with no additional uv parametrization of the second sample position.

```

pixel fragmentShader(fragment, volume, isoValue, textureMap, light,
    textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue)
        light.diffuse = textureMap(computeUV(fragment.tex0))
        normal = voxel.xyz * 2.0 - 1.0 * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif

```

Figure 5.4: OOP 2D texturing fragment shader

```

pixel fragmentShader(fragment, volume, weight, textureMap, light,
    textureMatrix)
    voxelF = volume(fragment.tex0)
    voxelB = volume(fragment.tex1)
    wght = weight(voxelF.a, voxelB.a)
    if (wght > 0.0)
        pos = lerp(IN.tex0, IN.tex1, wght);
        normal = (lerp(voxelF.xyz, voxelB.a, wght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        light.diffuse = textureMap(computeUV(pos))
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif

```

Figure 5.5: OOP interpolated 2D texturing fragment shader

5.1.5 Image-order Single Pass 2D Texturing

The IOS method can provide an additional speed-up when considering slabs rather than slices. The entire ray sampled in a single fragment shader program and can thus benefit from the already known previous samples during ray traversal. Additionally it is possible to include the speed-up mechanisms of early ray termination, deferred shading and empty space leaping. Figure 5.6 is the fragment shader for 2D texturing with slice sampling which is adapted from figure 3.37 and figure 5.7 is the slab based counterpart which is adapted from figure 3.38.

Empty space leaping can be defined with an octree for standard volume datasets or distance values for distance field datasets. The octree method has the disadvantage of requiring each sample to firstly lookup an octree value. The octree cell cannot be skipped since the requirement for a complex intersection function must be realised which adds more instructions to the fragment shader. A decision is also required if querying the octree at every sample location to decide if a particular sample contributes to the final image. This is usually achieved by addressing a texture map which requires further texture fetches to a pre-computed lookup table. The distance field space skipping strategy is better suited to image-order techniques and can be included with a 1D lookup table and additional instruction when rendering distance field datasets. The previous definitions of empty space leaping strategies can be directly imported (see section 3.2.4)

```

pixel fragmentShader(fragment, volume, dir, isoValue, light, textureMap,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(rayPos)
        if (voxel.a > isoValue)
            break
        endif
        rayPos += direction
        .
        .    // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (voxel.a > isoValue)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        light.diffuse = textureMap(computeUV(raypos))
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 5.6: IOS 2D texturing fragment shader

```

pixel fragmentShader(fragment, volume, dir, weight, light, textureMap,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    voxel1 = volume(rayPos)
    rayPos += direction
    while (true)
        voxel2 = volume(rayPos)
        wght = weight(voxel1.a, voxel2.a).a;
        if (weight(voxel1.a, voxel2.a).a > 0.0)
            break
        endif
        voxel1 = voxel2
        rayPos += direction
        .
        .    // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (wght > 0.0)
        normal = (lerp(voxel1.xyz, voxel2.xyz, wght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        rayPos = lerp (rayPos - direction, rayPos, wght)
        light.diffuse = textureMap(computeUV(rayPos))
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 5.7: IOS interpolated 2D texturing fragment shader

5.1.6 Results

Two algorithms are explored for 2D texturing volume objects, the slice sampling and slab sampling strategies. Additionally the IOS rendering strategy is used to include the acceleration techniques of early ray termination, empty space leaping and deferred shading. Performance measurements are based on sphere mapping computed in the fragment shader without the additional speed-up technique of pre-computing the uv parametrization, the texture map is 512×256 with 3 channels for $\langle r, g, b \rangle$. The additional alpha channel is not required since no blending is to be computed, however a semi-transparent extension of this algorithm can use the alpha channel to highlight certain regions of the texture map. The performance of this algorithm is considered constant across differing texture maps as complexity is affected by the underlying object only.

Figure 5.8 shows the results of the OOP techniques for 2D texturing applied to the *BuckyBall* dataset, figure 5.9 shows the results when applied to the *CTHeadDist* dataset which is more complex and has been shown to require additional sampling to accurately reconstruct an iso-surface. Additionally the *CTHeadDist* dataset is rendered with empty space leaping for the IOS approaches.

Table 5.1 shows the performance achieved with each algorithm for 2D texturing of volume objects. Figures 5.8 and 5.9 provide examples of the image quality achieved with 2D texturing algorithms and the respective sampling frequency. The *BuckyBall* dataset (see Figure 5.8) images are not shaded and shaded for comparison of sampling. The consideration of two samples with interpolation for every sampling point clearly generates better image quality for the same sampling frequency with single sample point consideration. The IOS method benefits greatly from deferred shading, empty space leaping and early ray termination.

Since the *computeUV()* function contains many instructions that cannot be vectorised in order to aid throughput, this proves to be expensive. Each frame rate was measured using conditional branching which in this case offered an improvement to the brute force method. A careful selection of which conditional branches to take dynamically is important since most of the conditionals in the IOS method are used to break out of the ray stepping process and only contain one instruction. It is more beneficial to allow the condition code mechanism to evaluate these conditionals since it is cheaper than computing a branch. The image-order technique provides the best scalable approach since it can utilise acceleration techniques.

5.2 Tangent Space

Since volume datasets do not exhibit a parametrically defined surface, no localised tangent space can be computed at each sample. By evaluating a proxy geometry at each sample point which is parametrically defined, an approximate tangent space can be derived. A sphere is used to represent these tangent vectors since from the origin of a co-ordinate system, unit length vectors all intersect the unit sphere exactly, thus every point on the unit sphere's surface defines a differing unit length vector.

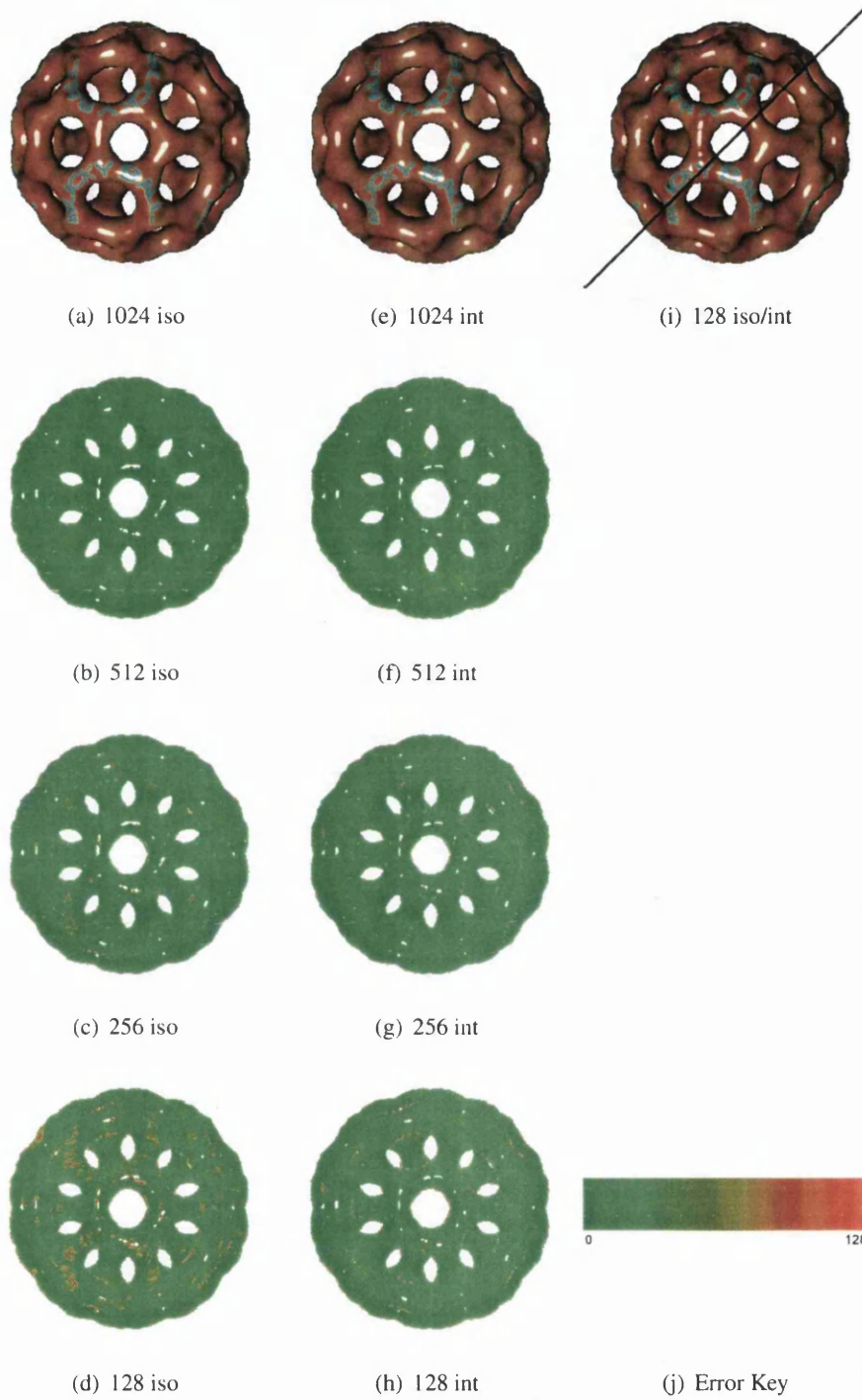


Figure 5.8: BuckyBall dataset 2D texture mapping (a) to (d) and interpolated 2D texture mapping (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared when subject to shading contributions to highlight sampling differences and the error range for difference images is given in (j).

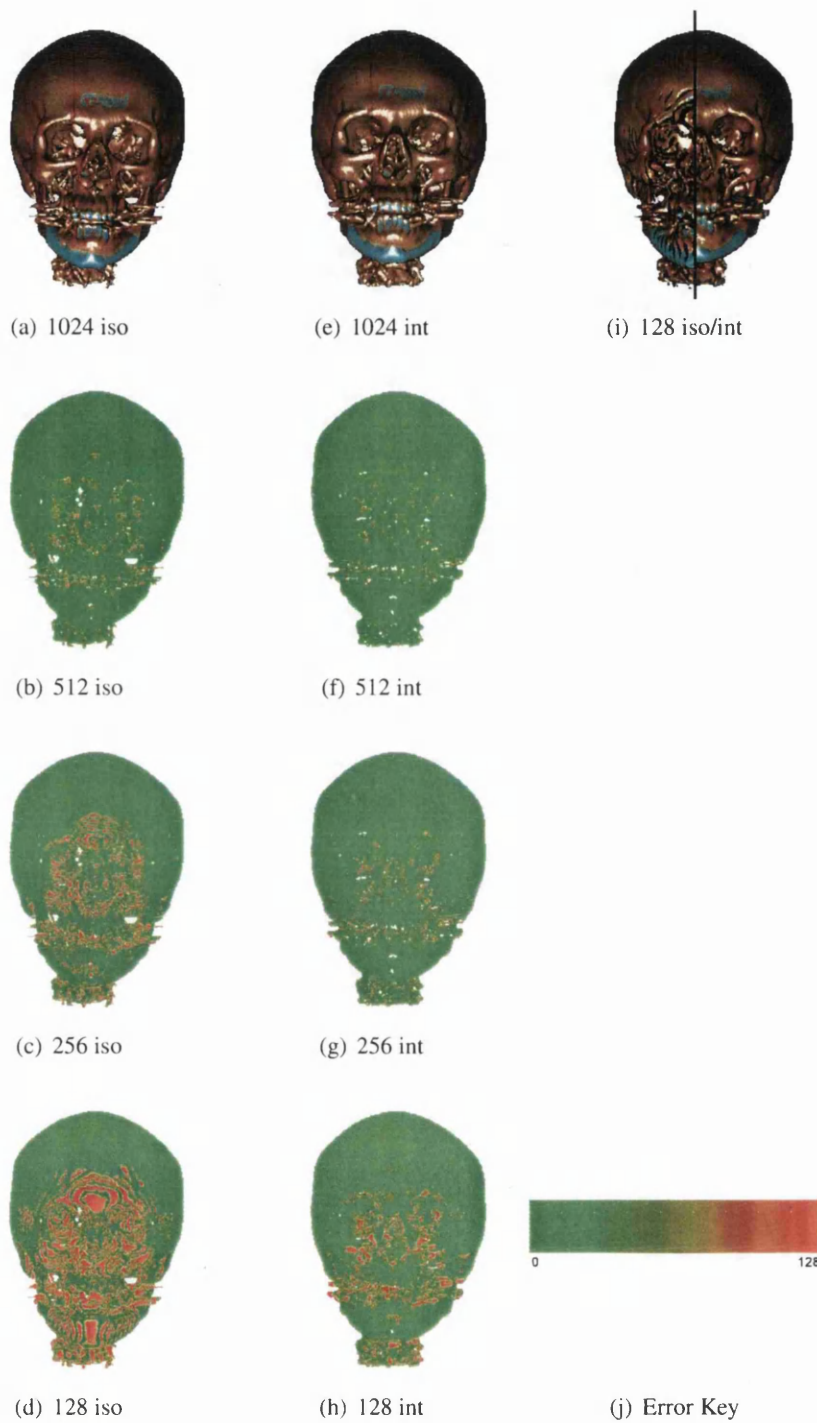


Figure 5.9: *CTHeadDist* dataset 2D texture mapping (a) to (d) and interpolated 2D texture mapping (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

Dataset (size)	Viewport	Lit	Slices	OOP		IOS		IOS-ES	
				Iso	Int	Iso	Int	Iso	Int
<i>BuckyBall</i> (32 ³) see Figure 5.8	512 ²	No	128	36	25	50	40	n/a	n/a
			256	18	12	27	22	n/a	n/a
			512	9	6	18	12	n/a	n/a
			1024	4	3	9	7	n/a	n/a
		Yes	128	24	18	50	40	n/a	n/a
			256	12	9	27	22	n/a	n/a
			512	6	4	18	12	n/a	n/a
			1024	3	2	9	7	n/a	n/a
	1024 ²	No	128	9	7	9	7	n/a	n/a
			256	4	3	4	3	n/a	n/a
			512	2	1	2	1	n/a	n/a
			1024	1	x	1	x	n/a	n/a
		Yes	128	7	6	9	7	n/a	n/a
			256	3	3	4	3	n/a	n/a
			512	1	1	2	1	n/a	n/a
			1024	< 1	< 1	1	< 1	n/a	n/a
<i>CTHeadDist</i> (256 ² × 128) see Figure 5.9	512 ²	No	128	32	28	30	20	42	27
			256	16	15	15	10	29	20
			512	8	7	7	5	17	14
			1024	4	4	3	2	9	7
		Yes	128	22	18	30	20	37	22
			256	11	8	15	10	25	18
			512	5	4	7	5	13	11
			1024	2	2	3	2	7	6
	1024 ²	No	128	8	7	14	12	16	9
			256	4	3	7	6	11	6
			512	2	1	3	3	7	4
			1024	1	x	1	1	4	2
		Yes	128	6	5	14	12	13	8
			256	3	2	7	6	8	5
			512	1	1	3	3	6	3
			1024	< 1	< 1	1	1	3	1

Table 5.1: 2D texture mapping frame rates in frames per second, Iso is single sample 2D texturing and Int is interpolated 2D texturing. Non lit variants do not compute any shading contributions whilst lit variants exhibit shading. All rates are rounded down.

It is known that a normal vector N is attributed with a tangent plane defined by two vectors T the tangent vector and B the binormal vector, however the normal vector is the cross product of tangent vectors (see Eqn 5.12)

$$N = T \times B \quad (5.12)$$

where $N \in [-1, 1]^3$, $T \in [-1, 1]^3$ and $B \in [-1, 1]^3$ are unit length vectors defined in 3D space.

By using a parametrically defined surface to compute the tangent and binormal vectors each normal vector will exhibit the same tangent and binormal vector. This allows each normal vectors tangent space to be consistent over an objects surface. Since this property is to be maintained, a computation of a local tangent space can be computed in respect of the gradient normal encountered at a sampling position and provide independence of position.

By assuming that the gradient normal is of unit length, a position on the unit sphere at the origin is obtained with the vector as the sphere's centre is the origin. This intersection with the sphere's surface can be described by the a azimuthal angle $\theta \in [0, 2\pi]$ and colatitude angle $\phi \in [0, \pi]$ (see Eqn 5.13).

$$\begin{aligned} \theta &= \arctan\left(\frac{N_x}{-N_z}\right) \\ \phi &= \arccos(N_y) \end{aligned} \quad (5.13)$$

An analogous to the projective texture mapping approach for reverse mapping is apparent since the same use of angles is defined to map a 3D point to 2D space. Instead of explicitly computing the coordinates in 2D space, the partial derivatives of the sphere function in respect of θ and ϕ are employed to derive the T and B vectors (see Eqn 5.14). This mapping is possible by observing that the θ angle changes with respect to the u axis and the ϕ angle changes with respect to the v axis accordingly.

$$\begin{aligned} T &= (\cos(\theta) \sin(\phi), 0, \sin(\phi) \sin(\theta)) \\ B &= (-\cos(\phi) \sin(\theta), \sin(\phi), \cos(\phi) \cos(\theta)) \end{aligned} \quad (5.14)$$

Since the volume dataset already defines a normal vector, only one of these vectors is required, a cross product can then be used to derive the other vector. Since T can return a zero length, B is computed and T is obtained with the cross product (see Eqn 5.2).

$$T = N \times B$$

These calculations allow a tangent, binormal, normal matrix T to be defined from the respective vectors (see Eqn 5.15).

$$T = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \quad (5.15)$$

A vector is moved from object space into tangent space upon multiplication with T . To move from tangent space into object space the inverse matrix T^{-1} is required. By observing the property that a unit sphere's surface describes all unit length vectors it is possible to pre-compute all possible vectors that correspond to a gradient normal. In practice an infinite representation is not available and the possible vectors must be quantised. Computing a 3D lookup table is not necessary to represent each vector since unit length vectors are used and positions within the lookup table that are not explicitly on the surface of the unit sphere provide no contribution. A 2D lookup table can be computed with a uv parametrization of the unit length normals components. This lookup table contains the binormal vector to create the T matrix. Tests have shown that the quality of output obtained in this method is not sufficient due to quantised normals being used and in addition a quantised output representation in 8 or 16 bits. Additionally a fast texture based uv parameterization has proven to introduce artifacts and therefore an expensive uv parameterization would be required. Future hardware that allows 32 bit floating point representations will allow an approximated acceleration by utilising pre-computed lookup tables.

5.3 Bump Mapping

Bump mapping is a well known flexible technique for adding fine detail to an object's surface. Generally highly complex surfaces are not possible in computer graphics without a highly detailed underlying object. Modelling techniques and data acquisition methods however do not usually provide such highly complex objects to render directly. In the event that such an object is defined, it will certainly be expensive to render. Many real world objects include bumps, wrinkles and grooves which are lost in building or scanning a model. Bump mapping facilitates an increasing need to import these small surface variations without impacting heavily on rendering expense. This section defines bump mapping and introduces the first efficient GPU implementations.

During the final stage of the pipeline an object's normal vectors are perturbed in a manner that affects the lighting calculation and results in a more wrinkled or bumpy surface. It has already been stated that providing colour alone in texture mapping algorithms is not always adequate for reproducing a highly realistic looking image. This method is capable of rendering increasingly complex surfaces without the requirement to increase the complexity or resolution of the underlying object.

Two techniques are introduced here and are optimised for the GPU pipeline. Both methods pre-compute the normal perturbation for fast application during rendering without the requirement of partial derivatives being evaluated during rendering. A normal map is utilised in both cases with pre-computed normal vectors in object or tangent space. The introduction of 3D bump mapping that fits the volumetric pipeline intrinsically is presented. This method takes advantage of 3D textures or volumes which are generally procedurally generated. This method however does not intrinsically model possible effects that require a painting on the surface of an object. Therefore the second method utilises the 2D texture mapping algorithms described in section 5.1 to form the basis of a 2D bump mapping technique.

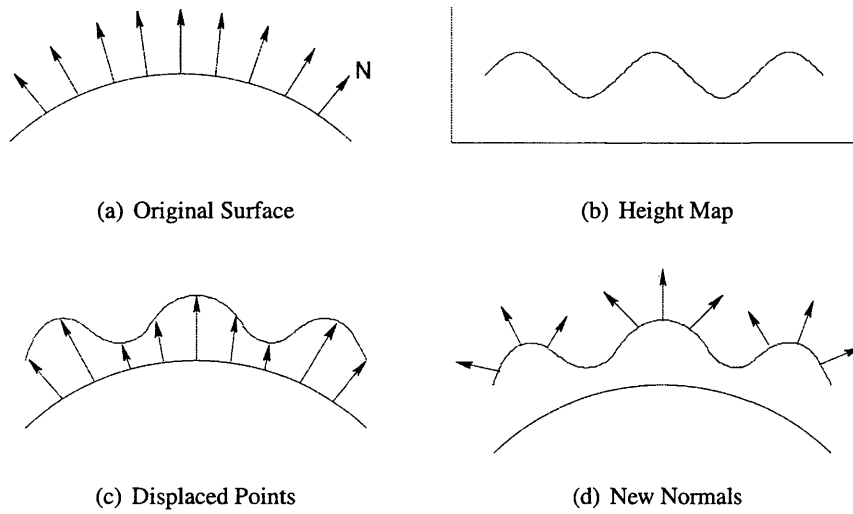


Figure 5.10: Bump mapping stages

Blinn [Bli78b] describes the original algorithm as performing a perturbation of the surface normal with respect to a height field. This height field is used to displace the surface along the surface normal to create the new wrinkled or bumpy surface definition. A further processing of the new localised surface is then used to derive new surface gradients for lighting calculations.

Satherley [Sat01] described bump mapping for volumetric objects in software. The original bump mapping function which computes perturbed normal vectors using partial derivatives is described and directly evaluated during ray traversal.

Partial derivatives of a parametric surface are used to firstly compute the surface normal N from the position $p \in \mathbb{R}^3$ (see Figure 5.10(a)). Two vectors are derived using the partial derivatives $P_u = (X_u, Y_u, Z_u)$ and $P_v = (X_v, Y_v, Z_v)$. These two vectors form a local co-ordinate system at p and are the tangent plane. The normal is defined with the cross product $N = P_u \times P_v$.

The surface position being processed is then displaced along the surface normal according to the height field F (see Figure 5.10(b)). This step of the algorithm does not explicitly move the surfaces geometry, instead acts as a step in the calculation (see Eqn 5.16).

$$P' = P + F \frac{N}{|N|} \quad (5.16)$$

This results in a virtual perturbed surface (see Figure 5.10(c)). The new surface normal is then taken from the partial derivatives of the new surface (see Eqn 5.17):

$$N' = \frac{P'_u}{P'_v} \quad (5.17)$$

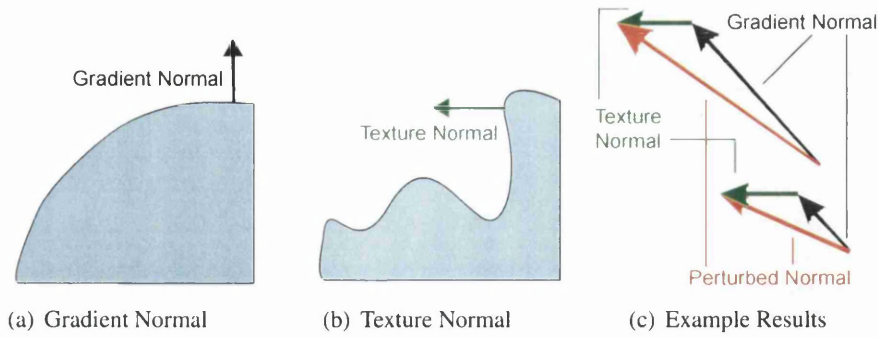


Figure 5.11: 3D Bump mapping normal perturbation. (c) represents perturbed normals according to discovered gradient normal(a) from the object and the normal discovered in the texture (b). Using differing magnitudes of the original gradient normal affects the final perturbed normals orientation.

Thus the perturbed normal is a function of the original parametric surface's partial derivatives and additionally the partial derivatives contained in the height field (see Figure 5.10(d)). The simplified derivation neglects F due to its small size and additionally defines D a local displacement factor. The full derivation is available in the original paper[Bli78b].

$$N' = N + D \quad (5.18)$$

$$D = F_u \left(\frac{N \times P_v}{|N|} \right) - F_v \left(\frac{N \times P_u}{|N|} \right)$$

5.3.1 3D Bump Mapping

This introduction of 3D bump mapping for volume datasets describes perturbing a gradient normal in a volume dataset before lighting takes place. The discussion here is restricted to iso-surfacing, however it is possible to apply the same functions to semi-transparent volume rendering. Previous iso-surfacing techniques have been described by applying the lighting calculations in eye space.

All volume gradients have been pre-processed and uploaded to the GPU hardware with the original scalar field in a 3D texture block. These gradient normals are contained in GPU memory in object space and therefore are not subject to any transformations. To correctly apply a lighting equation these gradient normals must first be transformed in some manner to eye space as the lighting position and screen space co-ordinate system is defined in this space.

The solid texture is then examined during iso-surfacing and the object's gradient normal is manipulated directly from the gradient defined in the solid texture. Previous explanations of procedural texturing (see section 4.2) noted the performance increase possible by pre-computing procedural texture primitives as texture blocks. These texture volumes can also be pre-processed to include gradient normals for evaluating additional techniques such as bump mapping.

Each object gradient normal defines a base vector for perturbation. The solid texture normal is added to the original gradient normal to arrive at a new position. The vector from the original point on the surface to the point defined by the contribution of vectors is the new perturbed gradient normal after a normalization step (see Eqn 5.19)

$$\text{bump}(N, V, k) = kN + V \quad (5.19)$$

where N is the original gradient normal, V is the gradient normal from the solid texture and k is a constant to describe bump severity.

The implementation of 3D bump mapping is efficient and can easily be added to a iso-surfacing fragment shader by simply adding two extra instructions. The bump map is encoded with object space gradient vectors and can be fetched with a single texture instruction. The offset can then be directly computed before the lighting stage of the pipeline is executed. Figure 5.12 provides the extension to the iso-surfacing shader of figure 3.25 and figure 5.13 is the direct replacement of figure 3.27 for slab sampling for the OOP rendering strategy. The slab lookup table must be pre-computed and updated upon iso-value changes. In practice the same performance measurements as standard iso-surfacing can be achieved with no dynamic conditional branches. The constant k used to scale the bump severity is implemented with a global variable and the texture map is presented as the function $\text{bumpMap}(\text{pos})$ where pos is the texture space co-ordinates.

```
pixel fragmentShader(fragment, volume, isoValue, bumpMap, light, k,
    textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue)
        bump = bumpMap(fragment.tex0) * 2.0 - 1.0
        normal = voxel.xyz * 2.0 - 1.0
        normal = normal * k + bump;
        normal = normal * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif
```

Figure 5.12: OOP 3D bump mapping fragment shader

```
pixel fragmentShader(fragment, volume, weight, bumpMap, light, k,
    textureMatrix)
    voxelf = volume(fragment.tex0)
    voxelb = volume(fragment.tex1)
    wght = weight(voxelf.a, voxelb.a)
    if (wght > 0.0)
        pos = lerp(fragment.tex0, fragment.tex1, wght)
        bump = bumpMap(pos) * 2.0 - 1.0;
        normal = lerp(voxelf.xyz, voxelb.a, wght)
        normal = normal * k + bump
        normal = normal * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif
```

Figure 5.13: OOP 3D interpolated bump mapping fragment shader


```

pixel fragmentShader(fragment, volume, isoValue, dir, light, bumpMap, k,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(rayPos)
        if (voxel.a > isoValue)
            break
        endif
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (voxel.a > isoValue)
        bump = bumpMap(rayPos) * 2.0 - 1.0
        normal = voxel.xyz * 2.0 - 1.0
        normal = normal * k + bump
        normal = normal * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 5.14: IOS 3D bump mapping fragment shader

The IOS technique allows the inclusion of early ray termination, empty space leaping and deferred shading. Figure 5.14 shows the slice sampling fragment shader which is adapted from figure 5.15 and figure 3.41 shows the slab sampling fragment shader which is adapted from 3.42. Branching can be set to non dynamic for the body of the ray casting loop since each conditional contains few instructions and it is faster to perform all of the outcomes and set condition codes rather than to take a branch. Empty space leaping can be incorporated using the functions presented in section 3.2.4. It has proven more efficient to pre-compute a 1D texture with quantised leaping values and multiply the normalised ray direction vector with the values contained in this table. The empty space leaping method is only well defined for distance field datasets since they encode euclidean distance values to the encoded iso-surface. Octree structures can be utilised in this approach for standard datasets, however every ray sample must firstly analyse the octree to determine if the sample contributes to the final image which is not efficient and can introduce additional overheads in many cases.

5.3.2 3D Bump Mapping Results

Both slice and slab sampling are considered for 3D bump mapping. Figure 5.16 shows the images obtained from the *BuckyBall* dataset at differing sampling frequencies and figure 5.17 shows the images obtained from the more complex *CTHeadDist* dataset. Timings are taken with no volume rotation to provide a performance comparison that is not affected by cache misses in 3D texturing hardware and to ensure the amount of fragments being processed are uniform throughout the tests. The bump map used is a texture lookup table of noise (see section 4.1) with the addition of a pre-computed gradient normal map using central differences.

```

pixel fragmentShader(fragment, volume, weight, dir, light, bumpMap, k,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    voxelF = volume(rayPos)
    rayPos += direction
    while (true)
        voxelB = volume(rayPos)
        wght = weight(voxelF.a, voxelB.a).a;
        if (wght > 0.0)
            break
        endif
        voxelF = voxelB;
        rayPos += direction
        .
        .    // further samples
        .
    if (direction.a < length(rayPos - fragment.tex0)) {
        break
    endif
endwhile
if (wght > 0.0)
    pos = lerp(rayPos - direction, rayPos, wght)
    bump = bumpMap(pos) * 2.0 - 1.0
    normal = lerp(voxelF.xyz, voxelB.xyz, wght) * 2.0 - 1.0
    normal = normal * k + bump
    normal = normal * textureMatrix.inverseTranspose
    pixel = lighting(normal, fragment.tex1, light)
endif

```

Figure 5.15: IOS 3D interpolated bump mapping fragment shader

Dataset (size)	Viewport	Slices	OOP		IOS		IOS-ES	
			Iso	Int	Iso	Int	Iso	Int
<i>BuckyBall</i> (32 ³) see Figure 5.16	512 ²	128	28	20	56	40	n/a	n/a
		256	14	10	35	24	n/a	n/a
		512	7	5	20	14	n/a	n/a
		1024	3	2	10	7	n/a	n/a
	1024 ²	128	8	6	10	7	n/a	n/a
		256	4	3	5	4	n/a	n/a
		512	2	1	3	2	n/a	n/a
		1024	1	x	1	1	n/a	n/a
<i>CTHeadDist</i> (256 ² × 128) see Figure 5.17	512 ²	128	20	13	35	16	58	36
		256	10	6	17	9	46	27
		512	5	3	9	4	34	21
		1024	2	1	4	2	19	16
	1024 ²	128	8	5	6	5	18	13
		256	4	2	3	3	14	10
		512	2	1	2	1	10	8
		1024	1	< 1	1	< 1	7	5

Table 5.2: 3D bump mapping frame rates in frames per second, Iso is single sample 3D bump mapping and Int is interpolated 3D bump mapping. All rates are rounded down.

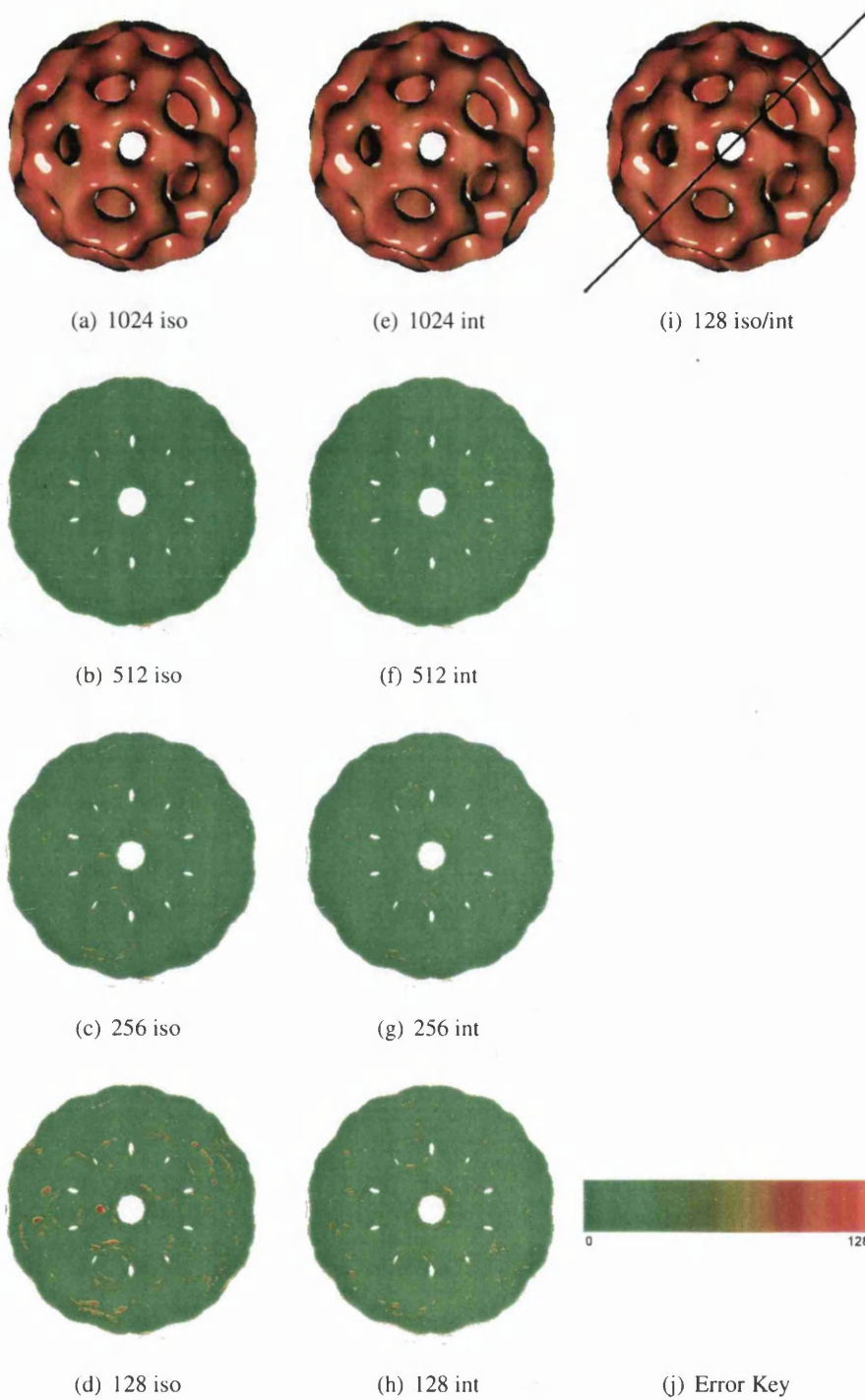


Figure 5.16: BuckyBall dataset 3D bump mapping (a) to (d) and interpolated 3D bump mapping (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

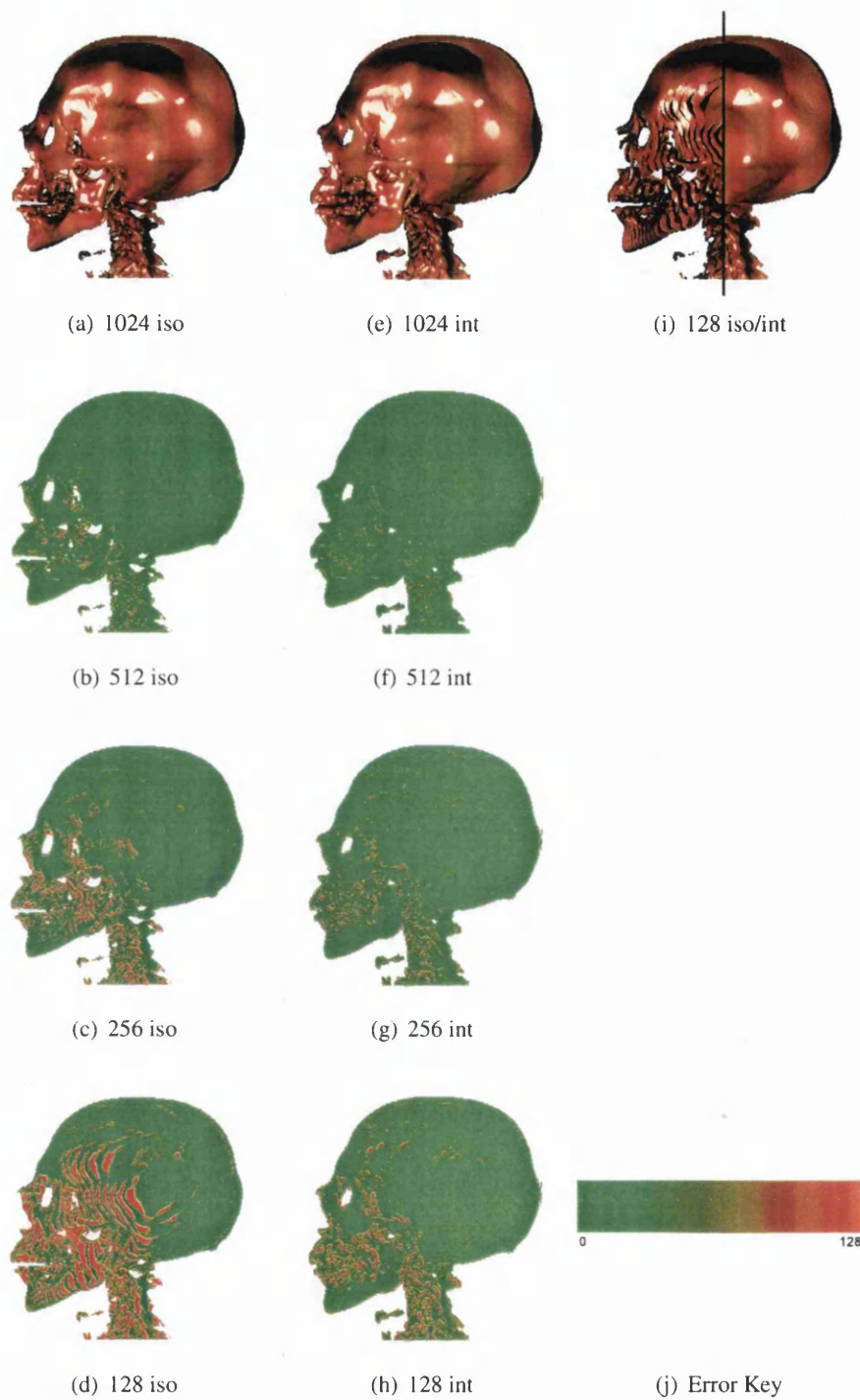


Figure 5.17: *CTHeadDist* dataset 3D bump mapping (a) to (d) and interpolated 3D bump mapping (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

Table 5.2 shows the performance measurements for 3D bump mapping. A parallel between the original iso-surface timings and the frame rates obtained here are evident. This is due to only two additional steps being required to successfully texture an object as the gradient normals from each source are in the same co-ordinate space. The IOS method allows high throughput since the conditionals evaluated during the casting of the ray are cheap to perform if dynamic branching is switched off. This method clearly benefits from the front-to-back ray casting strategy as well as the ability to perform early ray termination and deferred shading. The inclusion of space leaping shows increasing performance as the sampling frequency increases. This method therefore is very scalable over differing viewport sizes and sampling conditions. The original intention of this algorithm was to provide a fast alternate bump mapping where procedural synthesis is used. The relative speed compared to other approaches is due to no uv parametrisation having to be performed.

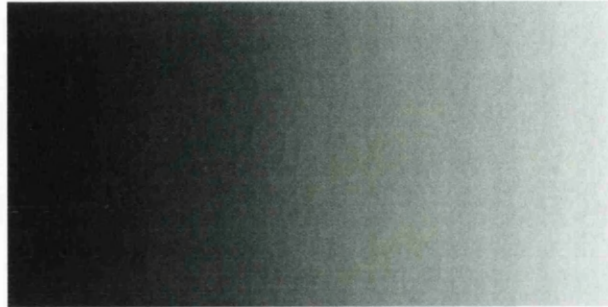
5.3.3 2D Bump mapping

The original bump mapping algorithm suffers from expensive computation using partial derivatives and additionally volumetric approaches exhibit a problem since a parametric representation must be used to derive the tangent space. A 2D function describes the bumpy surface and requires a uv parameterization of the objects surface to evaluate the function (see Figure 5.18(a)). Projective texturing techniques are used for this purpose in volumetric approaches since no direct uv parameterization is available (see section 5.1). The expensive computation can be reduced by pre-computing gradients for a bump function by observing local differences (see section 2.4) in two dimensions (see Figure 5.18(b)). The bump function can be procedurally synthesised or the result of a texture artist painting a monochrome 2D height field. The normal map is generally computed in respect of a template defining the object being rendered, or additionally in respect of the mapping function being used. The latter approach allows reuse of bump textures over different objects. The result of the computation is the normal vectors which can be represented with three colour channels $< r, g, b >$ in a texture map, additionally the height field is stored in the α channel.

These pre-computed normal vectors can then directly replace the original gradient vectors encountered at surface positions. This pre-computation has no underlying object definition and therefore the vectors are not correctly oriented in respect of the object being rendered. The pre-computed normal vectors are held in tangent space and require a mapping into object space (see section 5.2).

During ray traversal an expensive call to the uv parameterization function must take place. As previously stated it is possible to compute a lookup table to include a speed-up for this function, however the current precision of texture formats introduce aliasing. Since this function is expensive to compute the fragment shaders presented here perform best when dynamic branches are computed. The fragment shader is assumed to have a *computeUV(pos)* and *computeBinormal(norm)* functions available as depicted in sections 5.1 and 5.2. Figure 5.19 adds 2D bump mapping to the iso-surfacing technique from figure 3.25 for OOP rendering. Figure 5.20 adds 2D bump mapping to the slab sampled iso-surfacing technique from figure 3.27 for OOP rendering.

In IOS rendering, the fragment shaders for computing 2D bump mapping are given in figures



(a) Height Map



(b) Normal Map



(c) Bump Mapping



(d) Displacement Mapping

Figure 5.18: Normal mapping for 2D bump mapping and 2D displacement mapping

```
pixel fragmentShader(fragment, volume, isoValue, light, bumpMap, k,
    textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue)
        bump = bumpMap(computeUV(fragment.tex0)) * 2.0 - 1.0
        normal = voxel.xyz * 2.0 - 1.0
        binormal = computeBinormal(normal)
        tbn[0] = cross(binormal, normal)
        tbn[1] = binormal
        tbn[2] = normal
        normal = (bump * tbn) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif
```

Figure 5.19: OOP 2D bump mapping fragment shader

```
pixel fragmentShader(fragment, volume, weight, bumpMap, light, k, textureMatrix)
    voxel0 = volume(fragment.tex0)
    voxel1 = volume(fragment.tex1)
    wght = weight(voxel0.a, voxel1.a)
    if (wght > 0.0)
        pos = lerp(fragment.tex0, fragment.tex1, wght)
        bump = bumpMap(computeUV(pos)) * 2.0 - 1.0;
        normal = lerp(voxel0.xyz, voxel1.a, wght) * 2.0 - 1.0
        binormal = computeBinormal(normal)
        tbn[0] = cross(binormal, normal)
        tbn[1] = binormal
        tbn[2] = normal
        normal = (bump * tbn) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else
        discard
    endif
```

Figure 5.20: OOP 2D interpolated bump mapping fragment shader

```

pixel fragmentShader(fragment, volume, dir, isoValue, light, bumpMap, k,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(rayPos)
        if (voxel.a > isoValue)
            break
        endif
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (voxel.a > isoValue)
        bump = bumpMap(computeUV(rayPos)) * 2.0 - 1.0
        normal = voxel.xyz * 2.0 - 1.0
        binormal = computeBinormal(normal)
        tbn[0] = cross(binormal, normal)
        tbn[1] = binormal
        tbn[2] = normal
        normal = (bump * tbn) * mvMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 5.21: IOS 2D bump mapping fragment shader

5.21 and 5.22 for slice and slab sampling respectively. They are additions to the original iso-surfacing techniques described in figures 3.41 and 3.42. The functions *computeUV(pos)* and *computeBinormal(norm)* are assumed to be available to the fragment shader analogous to the OOP method. The IOS method benefits from the ability to perform empty space leaping, early ray termination and deferred shading. In practice the conditionals contained in the ray sampling loop can be performed in condition code mode since there are few instructions. Since the algorithm allows deferred shading, the computation of tangent space and the resulting matrix multiplication are performed only once, offering a substantial gain in throughput. The benefits of early ray termination on iso-surface intersection and empty space leaping can also be performed with the IOS approach. Empty space leaping is described for distance field datasets (see section 3.2.4), standard datasets require an octree structure which is inefficient in image order approaches on GPU hardware since each sample along the ray must query the octree to ascertain if it will contribute to the final image. In addition the slab sampling method benefits from reuse of previous results to reduce texture fetch overheads. This reuse of previous samples cannot be used with empty space leaping since the increment between sample stages is not guaranteed.

5.3.4 2D Bump Mapping Results

Table 5.3 gives the results of the 2D bump mapping algorithm. This algorithm is an accelerated version of the original algorithm due to pre-computing the tangent space normals. This algorithm is more expensive than its 3D bump mapping counterpart since an additional matrix multiplication must be performed to allow tangent space normal vectors into


```

pixel fragmentShader(fragment, volume, dir, weight, light, bumpMap, k,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    voxelF = volume(rayPos)
    rayPos += direction
    while (true)
        voxelB = volume(rayPos)
        wght = weight(voxelF.a, voxelB.a).a;
        if (wght > 0.0)
            break
        endif
        voxelF = voxelB;
        rayPos += direction
        .
        // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (wght > 0.0)
        normal = lerp(voxelF.xyz, voxelB.xyz, wght) * 2.0 - 1.0
        pos = lerp(rayPos - direction, rayPos, wght)
        bump = bumpMap(computeUV(pos)) * 2.0 - 1.0
        binormal = computeBinormal(normal)
        tbn[0] = cross(binormal, normal)
        tbn[1] = binormal
        tbn[2] = normal
        normal = (bump * tbn) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex1, light)
    endif

```

Figure 5.22: IOS 2D interpolated bump mapping fragment shader

Dataset (size)	Viewport	Slices	OOP		IOS		IOS-ES	
			Iso	Int	Iso	Int	Iso	Int
<i>BuckyBall</i> (32 ³) see Figure 5.23	512 ²	128	14	12	55	40	n/a	n/a
		256	7	6	35	24	n/a	n/a
		512	3	3	20	14	n/a	n/a
		1024	1	1	10	7	n/a	n/a
	1024 ²	128	4	4	10	8	n/a	n/a
		256	2	2	5	4	n/a	n/a
		512	1	1	3	2	n/a	n/a
		1024	< 1	< 1	1	1	n/a	n/a
<i>CTHeadDist</i> (256 ² × 128) see Figure 5.24	512 ²	128	12	9	37	26	42	28
		256	6	4	19	15	32	19
		512	3	2	10	8	21	12
		1024	1	1	5	4	14	8
	1024 ²	128	5	4	7	5	16	13
		256	2	2	4	3	12	10
		512	1	1	2	1	9	7
		1024	< 1	< 1	1	< 1	5	4

Table 5.3: 2D bump mapping frame rates in frames per second, Iso is single sample 2D bump mapping and Int is interpolated 2D bump mapping. All rates are rounded down.



Figure 5.23: BuckyBall dataset 2D bump mapping (a) to (d) and interpolated 2D bump mapping (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

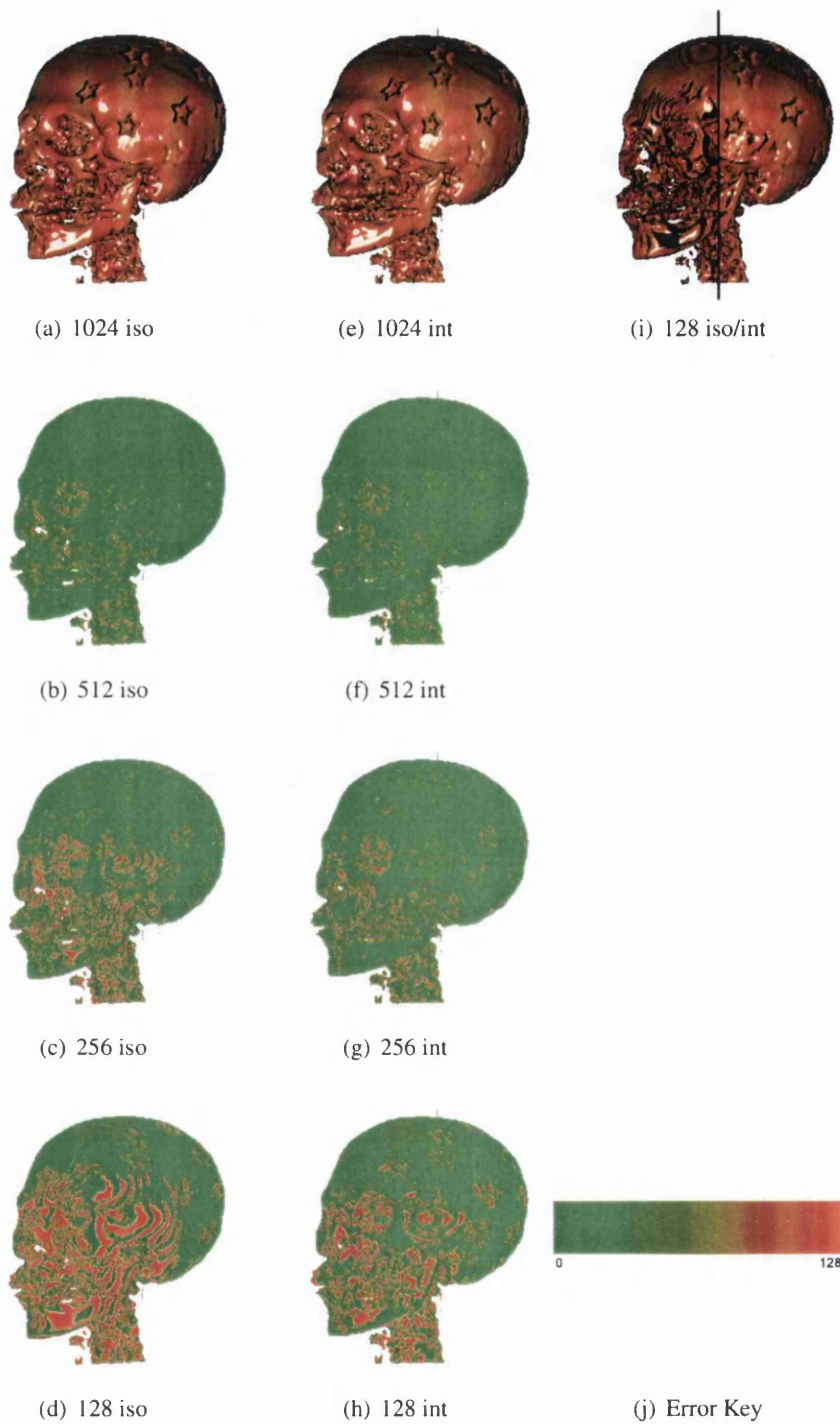


Figure 5.24: *CTHeadDist* dataset 2D bump mapping (a) to (d) and interpolated 2D bump mapping (e) to (h) images rendered into a 512^2 viewport with differing sample frequencies. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

the object space. An additional speed-up is possible based upon object space lighting. This would reduce the amount of matrix multiplication operations throughout the pipeline since no mapping is required for moving from object space into eye space. This can be performed by changing all the lighting conditions into object space firstly. An additional performance gain might be achieved by firstly encoding each of the tangent space normals in the bump map into object space. A pre-computation pass would be necessary and would require a template (forward mapping) rendering of the object's surface to deliver the gradient normals, and secondly a computation pass to move tangent space replacement vectors into object space. Since the template rendering is a ray-casting problem, computing a template each frame would half the performance measurements, but could potentially be achieved for every iso-surface change at the loss of interactivity.

Figures 5.23 and 5.24 are example outputs from the two sampling methods. Slab sampling outperforms slice sampling in producing an image with less artifacts for the same sampling frequency and additionally benefits from an increased throughput due to a lower sampling frequency being required to obtain an accurate output image. The IOS method of deferred shading and empty space leaping for distance fields enables fast frame rates to be obtained since the OOP counterpart is performing a uv parametrisation for any scalar encountered above the iso-value as no deferred shading or early ray termination can be performed. Since this adds instructions to each sample where the encountered scalar is above the iso-value, a 40% overhead is incurred which slows the algorithm down significantly. Additionally the tangent space to object space matrix multiplication is performed for all scalars encountered that are larger than the iso-value. The use of dynamic conditional branches in this case to skip instructions containing the uv parametrisation and matrix multiplication also introduces more overhead. Pre-computing the uv parametrisation will aid these problems by removing many instructions with the expense of an extra texture operation. In addition pre-computed object space normal maps or lighting in object space will accelerate the OOP method.

5.4 Displacement Mapping

New displacement mapping techniques for volume objects are introduced in this section to the GPU volume rendering pipeline. The goal of displacement mapping in volume graphics is two-fold, the first important feature is the ability to define volume objects in a modelling environment by utilising a simple object definition that is a 3D space function. The second feature is the ability to create realistic surface properties of arbitrary objects for more simple volume representations. Many more complex objects can be defined by applying a displacement map to an existing surface. This section introduces volume displacement mapping on GPU hardware and additionally introduces a new displacement mapping technique which removes the restrictions of defining displaced surface properties from an original surface and includes the ability to define mesostructure around the volume object which can be disjoint from the original surface.

Cook [Coo84] originally described displacement mapping for surface graphics with the use of a height map. Displacements were defined by moving vertices along their surface normal

vectors. This can be a complex problem, either the entire mesh has to be reprocessed or additional geometry has to be added to represent the displacement map where the original surface does not contain enough vertices to accurately account for the surface deformation.

Hirche *et al.* [HEGD04] and Porumbescu *et al.* [PBFJ05] used proxy geometry to represent a displacement region around an object for surfaces for GPU hardware implementation. The level of sampling required is dependant on the detail contained in the displacement map and therefore exhibits a geometric complexity. Both methods describe a height field.

Wang *et al.* [WWT⁺03, WTL⁺04] remove this restrictive geometric complexity by computing the displacement from the original object's surface on GPU hardware. However heavy pre-computation is required to form a large lookup table. The first method outlined was view dependant which required firstly sampling a higher resolution model at differing viewing angles for 2D height field approaches. The subsequent model is not view dependent, but requires a large pre-processed look up table. Additionally the notion of mesostructure is applied to displacement maps that removes previous restrictions of displacements having to emanate from the original surface.

Winter [Win02] describes 2D displacement mapping in software from height maps. A distance field dataset is used to describe a displacement region where the height map is evaluated during rendering.

Displacement mapping in volume graphics can be introduced with an overhead that does not rise with object complexity and therefore offers a more efficient alternative. This allows a complex surface representation to be voxelised into a distance field and displacement mapped for further detail. This step is also a useful tool at modelling and voxelisation stages when importing a surface based object that includes a surface description as a height map. It can also be used as a generalized modelling tool for deforming the surface characteristics of an underlying object. Since the displacement region can be large, many objects are more intuitive to create in this manner.

Distance field volume datasets are used to enable a segmentation of the dataset into three regions. These regions represent the outside of the object, the interior of the object and additionally the region for applying the displacement map. This allows the size of the region of displacement to be controlled which allows a great deal of control over the displacement. The segmentation is performed analogously to the previous description of hypertexturing by using an object density function (see Eqn 5.20). An alteration to the distribution of the displacement region is performed to ensure that values increase away from the original surface to represent height.

$$D(p) = \begin{cases} 1.0 & \text{if } |p| \leq r_i^2 \\ 0.0 & \text{if } |p| \geq r_o^2 \\ 1.0 - \frac{|p| - r_o}{r_i - r_o} & \text{otherwise.} \end{cases} \quad (5.20)$$

where r_i is the inner distance or iso-surface, r_o is the outer distance or soft region boundary and $|x|$ represents the distance field value. 1 is returned for samples inside the object and are subject to iso-surfacing, 0 is returned for samples outside the object and soft-region boundary and $1.0 - \frac{|p| - r_o}{r_i - r_o}$ the distance from the surface is returned when samples are between the

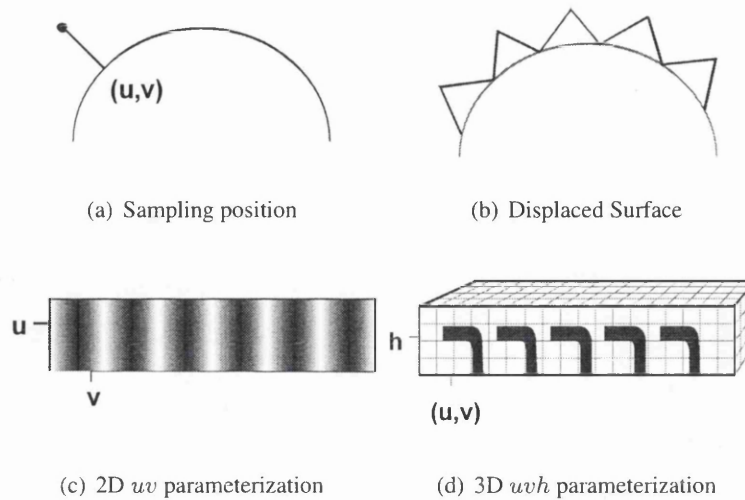


Figure 5.25: 2D and 3D displacement mapping strategies

soft-region boundary and surface boundary.

Displacement mapping is essentially an extension that overcomes limitations with the bump mapping algorithm. Bump mapping suffers from not being able to describe a displaced surface around the silhouette of an object. Additionally the magnitude of the bumps is limited since the curvature of the surface can obscure the lighting under rotations where the normal is not pointing directly towards the light source. Displacement mapping does combat both of these shortcomings since the underlying object geometry is altered. Two methods of implementing displacement mapping are presented, a 2D height map method analogous to bump mapping with 2D bump maps and additionally a 3D volume displacement mapping method that exhibits greater flexibility and modelling potential.

5.4.1 2D Displacement Mapping

The 2D displacement mapping method is an extension to bump mapping and can use height fields employed in bump mapping directly. Additionally the gradients can be pre-computed as with bump mapping and stored in tangent space in the same manner. Whilst bump mapping directly replaces the objects normal vectors, displacement mapping additionally requires the original height field to enable computing the distance to move from the original surface.

A segmentation is performed to create a displacement region (see Eqn 5.20). During ray traversal the displacement region is evaluated with respect to the 2D height field. Firstly a uv parameterization of the objects surface is required and one of the 2D texturing intermediate surface geometries is used to closely approximate the underlying object (see section 5.1). The computation of tangent space is also required to transform pre-computed normal vectors from tangent space into object space.

These functions described previously are assumed to be present as functions for use in the fragment shader. A function $2dDisplace(voxel, pos, dispMap)$ is introduced into the available fragment shader functions to compute the displacement where $voxel$ contains the gradient normal and distance value from the sample position, pos is the sampling position and $dispMap$ is the 2D displacement texture map.

For each sample encountered along a ray that intersects the displacement region, a displacement is computed by traversing along the reverse direction of the gradient vector to the original surface (see Figure 5.25(a)). Using a distance field, this can be achieved without the need for tracing a further ray since the distance function describes the distance to the surface. At the iso-surface location a uv parameterization is computed from the position of the surface. This uv parametrization is used to address the displacement map lookup table which returns a $\langle r, g, b, \alpha \rangle$ containing a tangent space normal vector and the height value (see Figure 5.25(c)). The height value is then compared with the original normalized distance encountered at the original sample location obtained from the object density function (see Eqn 5.20). A segmentation is then performed with the object density normalized distance and height field value to denote inclusion or exclusion from the displaced surface (see Figure 5.25(b) and Eqn 5.21).

$$2dDisplace(N, p, D(p), T) = \begin{cases} 1 & \text{if } T(uv(p')) < D(d) \\ 0 & \text{otherwise} \end{cases} \quad (5.21)$$

where p is the sample point, $p' = p + (-Nd)$ is the point on the original surface along the reversed gradient normal N with length d the distance value encountered and $T(p)$ is the texel located in the displacement map for mapped uv co-ordinates from p' .

The fragment shader for 2D displacement mapping using the OOP rendering technique is given in figure 5.26. Slab rendering cannot be employed for 2D texture mapping since there is no iso-value to consider. Heights are instead used to define the surface of an object and the evaluation of two distinct samples within the displacement region does not guarantee a linear progression. A possible extension to this algorithm is to pre-compute the height field into a scalar volume dataset or distance field which would allow a slab sampling strategy to be used with the 3D displacement mapping algorithm.

IOS2D displacement mapping is depicted in figure 5.27. This method benefits from the ability to perform early ray termination, empty space leaping and deferred shading. Since the displacement region must be evaluated for every sample encountered as a height field must be queried, additional lookups are required compared to the bump mapping algorithm counterpart. This highlights the need for efficient ray termination strategies as the fragment program generated will contain many more operations per sample. Additionally each encountered displacement region iso-surface intersection requires the computation of tangent space and additionally a matrix multiplication for the tangent space to object space mapping. The iso-surface decision must be made at each sample inside the soft region and therefore a uv parametrisation must be made for each sample. The empty space skipping method also must take into account the displacement region boundary. Therefore an adjustment is required to correctly skip up to the displacement region. This can be achieved with the previously defined functions 3.3 or 3.4 by adjusting the iso-value to be the displacement boundary value.

```

pixel fragmentShader(fragment, volume, dispMap, isoValue, light,
    textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue.a)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (voxel.a < isoValue.a and voxel.a > isoValue.z)
        h = 1.0 - ((voxel.a - isoValue.z) / (isoValue.a - isoValue.z))
        normal = voxel.xyz * 2.0 - 1.0
        disp = 2dDisplace(normal, fragment.tex0, h, dispMap)
        if (disp.a > 0.0)
            binormal = computeBinormal(normal)
            tbn[0] = cross(binormal, normal)
            tbn[1] = binormal
            tbn[2] = normal
            normal = ((disp.xyz * 2.0 - 1.0) * tbn) *
                textureMatrix.inverseTranspose
            pixel = lighting(normal, fragment.tex2, light)
        endif
    else
        discard
    endif

```

Figure 5.26: OOP 2D displacement mapping fragment shader

5.4.2 2D Displacement Mapping Results

Table 5.4 shows the performance characteristics of 2D displacement mapping. The extra burden on the fragment shader stage of the pipeline required for sampling each displacement region sample with a texture lookup affects the performance drastically in comparison to bump mapping. Including the acceleration methods outlined still produces much slower results. However this is due to the extra samples contributing the surface which have to be created dynamically. The rendered results are far superior in quality compared to 2D bump mapping. It is expected that future hardware will include an improved branching mechanism which will allow these algorithms to produce a higher throughput. The dynamic branching is utilised to avoid expensive operations in the displacement region, however introduces a large overhead. Sharp edges prove to be difficult to render for this technique since the underlying 2D height field might define an edge that is no more than a pixel wide in the original height field. The previously mentioned strategy to include slab rendering techniques will eliminate this problem. In this case, this algorithm has the benefit of providing arbitrary large displacements and can be used as a modelling approach to creating solid volumetric objects with a 2D definition applied to a 3D object. This can be more intuitive than existing methods such as combining several volume datasets and includes the benefit of keeping the original object and resulting objects definition separate.

5.4.3 Volume Displacement Mapping

Volume displacement mapping is an extension of the 2D displacement mapping algorithm that allows the inclusion of volumetric datasets that contain *mesostructure*. This new method removes a restriction of the 2D displacement mapping algorithm which requires that all displacements emanate from the original surface. The 3D displacement mapping extension


```

pixel fragmentShader(fragment, volume, dir, isoValue, light, dispMap,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(rayPos)
        if (voxel.a > isoValue.a)
            break
        isoValue.x = 1.0
        else if (voxel.a <= isoValue.a and voxel.a >= isoValue.z)
            break
        isoValue.x = 0.5
        endif
        rayPos += direction
        .
        . // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (isoValue.x = 1.0)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (isoValue.x = 0.5)
        h = 1.0 - ((voxel.a - isoValue.z) / (isoValue.a - isoValue.z))
        normal = voxel.xyz * 2.0 - 1.0
        disp = 2dDisplace(normal, rayPos, h, dispMap)
        if (disp.a > 0.0)
            binormal = computeBinormal(normal)
            tbn[0] = cross(binormal, normal)
            tbn[1] = binormal
            tbn[2] = normal
            normal = ((disp.xyz * 2.0 - 1.0) * tbn) *
                textureMatrix.inverseTranspose
            pixel = lighting(normal, fragment.tex2, light)
        endif
    endif
endif

```

Figure 5.27: IOS 2D displacement mapping fragment shader

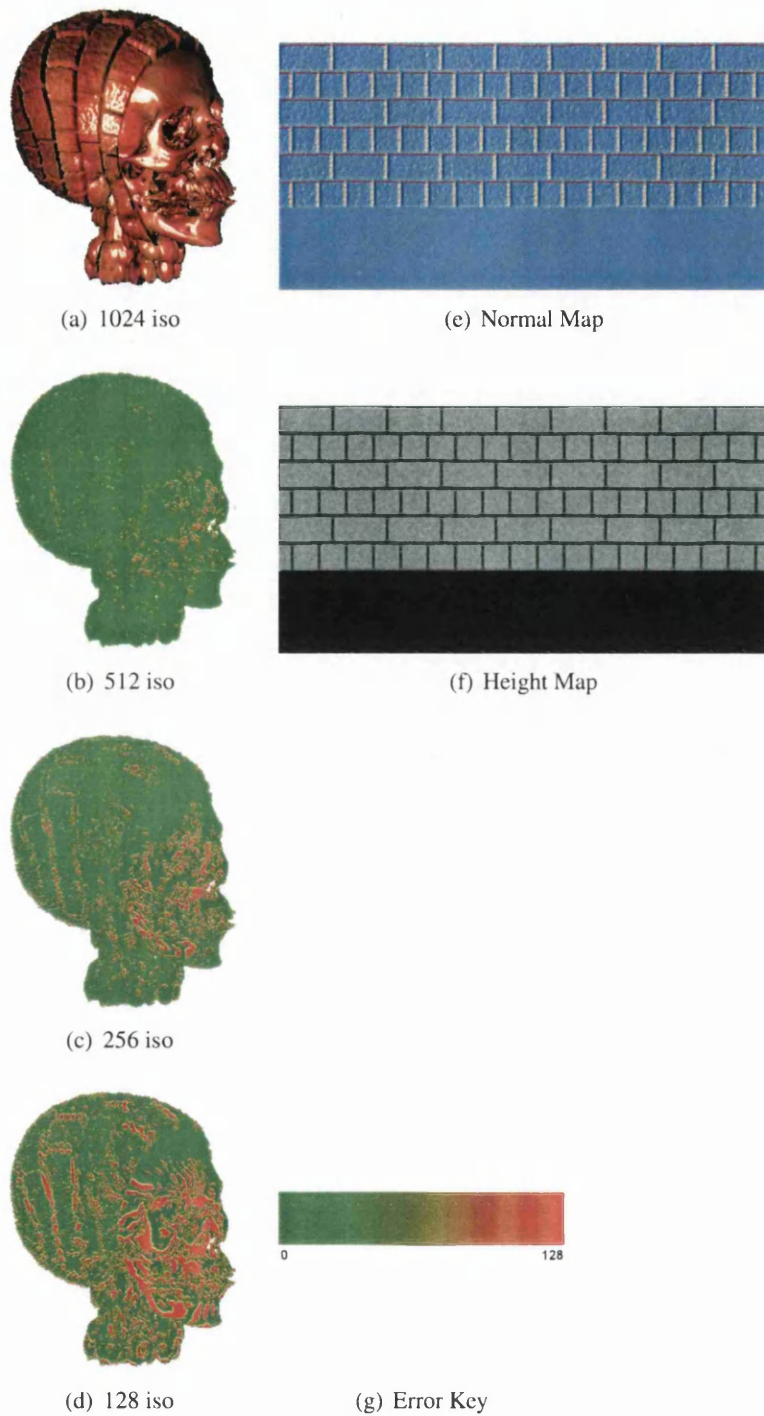


Figure 5.28: *CTHeadDist* dataset 2D displacement mapping images (a) to (d) rendered into a 512^2 viewport with differing sample frequencies from height map (e) and normal map (f). (b) to (d) are the difference images from (a) to visualize artefacts introduced with lower sampling rates and the error range for difference images is given in (h).

Dataset (size)	Viewport	Slices	OOP	IOS	IOS-ES
<i>CTHeadDist</i> (256 ² × 128) see Figure 5.28	512 ²	128	16	8	7
		256	7	4	5
		512	3	2	3
		1024	1	1	2
	1024 ²	128	6	3	2
		256	3	1	1
		512	1	< 1	< 1
		1024	< 1	< 1	< 1
<i>SphereDist</i> (256 ³)	512 ²	128	10	10	12
		256	5	5	8
		512	2	2	5
		1024	1	1	3
	1024 ²	128	4	4	6
		256	2	2	4
		512	1	1	2
		1024	< 1	< 1	1

Table 5.4: 2D displacement mapping frame rates in frames per second, *Iso* is single sample 2D displacement mapping and *Int* is interpolated 2D displacement mapping. All rates are rounded down.

removes this restriction and allows placement of any medium in the displacement region. The normalised height encountered in the displacement region is used to address a volume dataset with the 2D dimensional uv parameters having been already calculated. An object density function segments the object to encode three regions in the dataset, outside the object, the displacement region and the original iso-surface. The description is not limited to iso-surfacing as blending can be enabled and fuzzy classification with a transfer function can be computed in the displacement region.

The functions *computeUV* and *computeBinormal* are assumed to be present in each fragment shader. The *3dDisplace(voxel, pos, dispMap)* function outputs a scalar instead of binary value to allow iso-surfacing and direct volume rendering where *voxel* encodes the original distance value and gradient normal, *pos* is the sample position and *dispMap* is the volume dataset containing tangent space gradient normals and scalar density or distance field values. Eqn 5.22 is the modified displacement function.

$$3dDisplace(N, p, D(p), T) = T(uv(p'), D(p)) \quad (5.22)$$

where p is the sample point, $p' = p + (-Nd)$ is the point on the original surface along the reversed gradient normal N with length d the distance value encountered and $T(p)$ is the texel located in the displacement map for mapped uv co-ordinates from p' .

Figures 5.29 and 5.30 define the OOP rendering strategies for 3D displacement mapping with slice and slab sampling respectively. Figures 5.31 and 5.32 define the 3D displacement mapping fragment shaders for the IOS approach. These shaders compute a second segmentation based on iso-value and perform iso-surfacing of the displacement region. This

requires that all samples in the displacement region are subject to a uv parameterisation of the surface which proves to be slow. Dynamic conditional branches are used to skip costly operations which introduces an overhead to compute these branches. The benefit of early ray termination for the IOS approach is evident since when a displacement region iso-surface intersection occurs, the rest of the ray does not require sampling. The deferred shading in this case results in only one tangent space to object space mapping per ray. The empty space leaping strategy is performed in respect of the displacement region boundary instead of the iso-value for the original surface.

```

pixel fragmentShader(fragment, volume, isoValue, light, dispMap, textureMatrix)
    voxel = volume(fragment.tex0)
    if (voxel.a > isoValue.a)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (voxel.a < isoValue.a and voxel.a > isoValue.z)
        h = 1.0 - ((voxel.a - isoValue.z) / (isoValue.a - isoValue.z))
        normal = voxel.xyz * 2.0 - 1.0
        disp = 3dDisplace(normal, rayPos, h, dispMap)
        if (disp.a > isoValue.y)
            binormal = computeBinormal(normal)
            tbn[0] = cross(binormal, normal)
            tbn[1] = binormal
            tbn[2] = normal
            normal = ((disp * 2.0 - 1.0) * tbn) *
                textureMatrix.inverseTranspose
            pixel = lighting(normal, fragment.tex2, light)
        endif
    else
        discard
    endif

```

Figure 5.29: OOP volume displacement mapping fragment shader

5.4.4 Volume Displacement Mapping Results

Table 5.5 gives the performance measurements for the volume displacement mapping algorithm. Although there is a noticeable decrease in throughput from the 2D displacement mapping algorithm, this performance drop is small. Both algorithms are complex to compute on GPU hardware and it is expected that as shader conditional branching improves, these algorithms will increase in throughput to reach other algorithms presented in this thesis. Figure 5.33 details images obtained using this method. The spheres mapped around the original sphere function are formed from the *SphereMeso* dataset which contains many small spheres as a distance field. These spheres can be seen to be disjoint from the original surface which highlights the removal of previous restrictions by using this approach.

Both approaches suffer from the heavy tangent space mappings and uv parametrisations required by the algorithm. As previously stated, pre-computing lookup tables for these functions currently produces artefacts in the final image due to quantisation. The acceleration techniques for IOS have proven to remove some burden but ultimately suffer from the conditional branching costs associated with skipping unnecessary uv parametrisations.

This algorithm removes the restrictions associated with 2D displacement mapping with a

```

pixel fragmentShader(fragment, volume, weight, weightDisp, light, isoValue,
    dispMap, textureMatrix)
    voxelF = volume(fragment.tex0)
    voxelB = volume(fragment.tex1)
    wght = weight(voxelF.a, voxelB.a).a
    if (wght > 0.0)
        normal = (lerp(voxelF.xyz, voxelB.xyz, wght) * 2.0 - 1.0) *
            textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (voxel.a < isoValue.a and voxel.a > isoValue.z)
        hf = 1.0 - ((voxelF.a - isoValue.z) / (isoValue.a - isoValue.z))
        hb = 1.0 - ((voxelB.a - isoValue.z) / (isoValue.a - isoValue.z))
        dispF = 3dDisplace(voxelF.xyz * 2.0 - 1.0, fragment.tex0, hf, dispMap)
        dispB = 3dDisplace(voxelB.xyz * 2.0 - 1.0, fragment.tex1, hb, dispMap)
        wghtDisp = weightDisp(dispF.a, dispB.a, wght).a
        if (wghtDisp > 0.0)
            normal = lerp(voxelF.xyz, voxelB.xyz) * 2.0 - 1.0
            dispNormal = lerp(dispF.xyz, dispB.xyz) * 2.0 - 1.0
            binormal = computeBinormal(normal)
            tbn[0] = cross(binormal, normal)
            tbn[1] = binormal
            tbn[2] = normal
            normal = (dispNormal * tbn) * textureMatrix.inverseTranspose
            pixel = lighting(normal, fragment.tex2, light)
        endif
    else
        discard
    endif
endif

```

Figure 5.30: OOP volume interpolated displacement mapping fragment shader

Dataset (size)	Viewport	Slices	OOP		IOS		IOS-ES	
			Iso	Int	Iso	Int	Iso	Int
<i>CTHeadDist</i> ($256^2 \times 128$)	512 ²	128	7	6	12	10	12	8
		256	4	3	7	5	8	5
		512	2	1	3	2	5	3
		1024	1	< 1	1	1	3	1
	1024 ²	128	3	2	3	2	6	3
		256	1	1	1	1	4	2
		512	< 1	< 1	< 1	< 1	2	1
		1024	< 1	< 1	< 1	< 1	1	< 1
<i>SphereDist</i> (256^3) see Figure 5.33	512 ²	128	10	7	9	8	n/a	n/a
		256	5	3	5	4	n/a	n/a
		512	2	1	2	2	n/a	n/a
		1024	1	< 1	1	1	n/a	n/a
	1024 ²	128	3	2	3	2	n/a	n/a
		256	1	1	1	1	n/a	n/a
		512	< 1	< 1	< 1	< 1	n/a	n/a
		1024	< 1	< 1	< 1	< 1	n/a	n/a

Table 5.5: Volume displacement mapping frame rates in frames per second, Iso is single sample volume displacement mapping and Int is interpolated volume displacement mapping. All results are rounded down.

```

pixel fragmentShader(fragment, volume, dir, isoValue, light, bumpMap,
    textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    while (true)
        voxel = volume(rayPos)
        if (voxel.a > isoValue.a)
            isoValue.x = 1.0
            break
        else if (voxel.a <= isoValue.a and voxel.a >= isoValue.z)
            h = 1.0 - ((voxel.a - isoValue.z) / (isoValue.a - isoValue.z))
            disp = 3dDisplace(voxel.xyz * 2.0 - 1.0, fragment.tex0, h, dispMap)
            if (disp.a > isoValue.y)
                isoValue.x = 0.5
                break
            endif
        endif
        rayPos += direction
        .
        // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (isoValue.x = 1.0)
        normal = (voxel.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (isoValue.x = 0.5)
        normal = (voxel.xyz * 2.0 - 1.0)
        binormal = computeBinormal(normal)
        tbn[0] = cross(binormal, normal)
        tbn[1] = binormal
        tbn[2] = normal
        normal = ((disp.xyz * 2.0 - 1.0) * tbn) *
            textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    endif

```

Figure 5.31: IOS volume displacement mapping fragment shader

```

pixel fragmentShader(fragment, volume, dir, weight, dispWeight, isoValue, light,
    dispMap, textureMatrix)
    direction = dir(fragment.wpos)
    rayPos = fragment.tex0
    voxelF = volume(rayPos)
    rayPos += direction
    while (true)
        voxelB = volume(rayPos)
        wght = weight(voxelF.a, voxelB.a).a
        if (wght > 0.0)
            isoValue.x = 1.0
            break
        else if (voxelF.a <= isoValue.a and voxelB.a >= isoValue.z)
            hf = 1.0 - ((voxelF.a - isoValue.z) / (isoValue.a - isoValue.z))
            hb = 1.0 - ((voxelB.a - isoValue.z) / (isoValue.a - isoValue.z))
            dispF = 3dDisplace(voxelF.xyz * 2.0 - 1.0, fragment.tex0,
                hf, dispMap)
            dispB = 3dDisplace(voxelB.xyz * 2.0 - 1.0, fragment.tex1,
                hb, dispMap)
            wghtDisp = weightDisp(dispF.a, dispB.a).a
            if (wghtDisp > 0.0)
                isoValue.x = 0.5
                break
            endif
        endif
        voxelF = voxelB
        rayPos += direction
        .
        .    // further samples
        .
        if (direction.a < length(rayPos - fragment.tex0)) {
            break
        }
        endif
    endwhile
    if (isoValue.a = 1.0)
        normal = (voxelF.xyz * 2.0 - 1.0) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    else if (isoValue.a = 0.5)
        normal = lerp(voxelF.xyz, voxelB.xyz) * 2.0 - 1.0
        dispNormal = lerp(dispF.xyz, dispB.xyz) * 2.0 - 1.0
        binormal = computeBinormal(normal)
        tbn[0] = cross(binormal, normal)
        tbn[1] = binormal
        tbn[2] = normal
        normal = (dispNormal * tbn) * textureMatrix.inverseTranspose
        pixel = lighting(normal, fragment.tex2, light)
    endif

```

Figure 5.32: IOS volume interpolated displacement mapping fragment shader

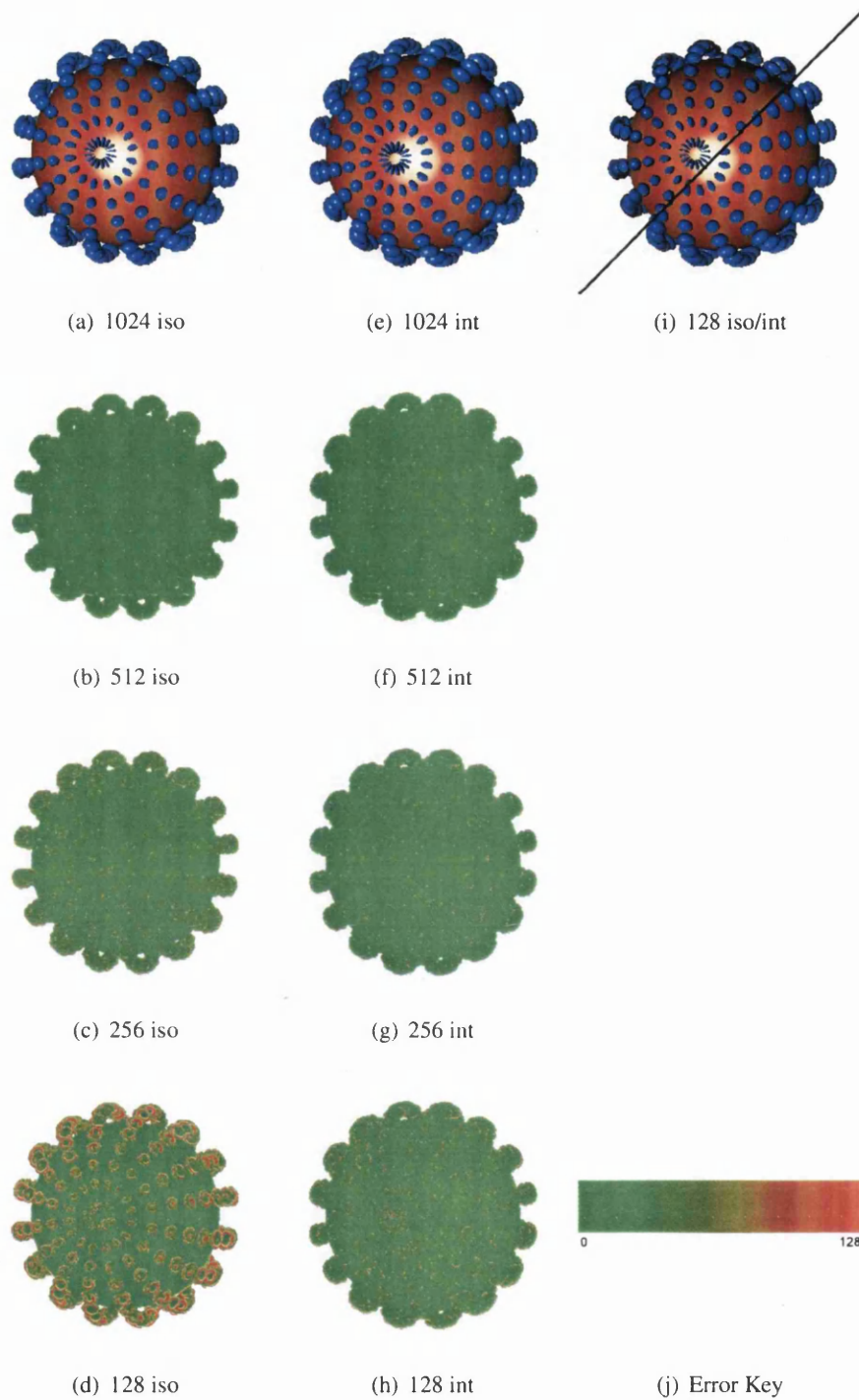


Figure 5.33: *SphereDist* dataset volume displacement mapping (a) to (d) and interpolated 3D displacement mapping (e) to (h) images rendered into a 512² viewport with differing sample frequencies. The displacement region is filled with the *SphereMesodataset*. Both (b) to (d) and (f) to (h) are the difference images from (a) and (e) respectively to visualize artefacts introduced with lower sampling rates. Both techniques (i) are compared to highlight sampling differences and the error range for difference images is given in (j).

similar overhead. Iso-surfacing has been described, but the rendering strategy for the displacement region can easily be adjusted to classify a density encountered in the displacement region and perform direct volume rendering with blending. In this case the algorithm is less expensive since no tangent space to object space mapping is required, and is similar to hypertexture with the addition of a uv parametrisation of the objects surface. This does allow many more effects to be rendered however since no procedural primitives are evaluated and any volume dataset can define the displacement region.

5.5 Summary

The algorithms defined in this chapter form the basis of a powerful descriptive environment for modelling general form with volume datasets. Both 2D texturing and bump mapping are important techniques to the surface graphics community to provide inexpensive realism in rendered scenes and objects. It has been shown that these methods fit well with the volume graphics pipeline with the addition to their real-time display. This allows surfaces to be imported into volumetric scenes with ease by voxelising original objects and using existing texture maps. Very complex surface meshes can benefit from this approach as general complexity in volume graphics remains constant for arbitrary complex datasets. The *CTHeadDist* dataset is shown here as requiring more sampling than other datasets, however interactive rates are maintained in most situations.

The introduction of displacement mapping into the volumetric pipeline provides a powerful tool for modelling objects and rendering finite details. The frame rates achieved so far for a complex algorithm are promising and in the future when dynamic branching hardware improves on GPU's these methods will show an increased throughput. Since the complexity of providing these tools is not much greater than the original problem, future hardware with better branch performance and increased bandwidth and clock speeds will allow these algorithms to run at over 10 fps, currently the branch costs are limiting the algorithm by introducing many additional cycles due to branching. In addition, more accurate texture maps and increased memory will allow the pre-computation of expensive functions such as the uv parametrisation and tangent space to object space mapping.

Chapter 6

Conclusion

Contents

6.1 Achievements	198
6.2 Further Work	198

The aim of this thesis was to provide a framework to compute volume graphics applications on commodity graphics hardware architectures. Particular focus has been extended towards texturing volume objects to provide realistic imagery with real world attributes. This work is seen as the initial steps in providing volume graphics to a wider audience and user base and to promote its use as a flexible graphical modelling and rendering representation, capable of superseding current surface representations. The explosion of programmable hardware onto the main consumer market will continually increase the possibilities to define more methods and strategies at an interactive rate.

The question of having to re-examine several established algorithms for use on the GPU is shown to be a nessicary step in providing real-time graphics solutions on single workstations. This is the most cost-effective manner to date to display real-time volume graphics and although it required revisiting algorithms designed at the inception of the subject, has enabled better understanding of these algorithms and allowed tighter implementations that are more efficient than their software cousins.

This work has raised questions regarding the implementations of GPU hardware and the future trends. The first issue is that of dynamic branching being so costly on current implementations. If this cost is reduced, many more complex applications can be implemented to take advantage of the parallel pipeline. Secondly the current debugging model is not sufficient enough to implement complex algorithms without a heavy development period with unknown problems. Most often a shader that does not work is simple to amend, however without being able to see registers and step though code, the GPU must be treated as a black box. Third, an implementation of a noise algorithm will significantly benefit many members of the graphics community and give truly real-time rich procedural textures. There are patents for specialised hardware that implement these algorithms but they are not currently included on the graphics hardware. This would be an enormous advantage to using

GPU pipelines in graphics rendering as procedurally generating noise without the specialist hardware is mostly too expensive to maintain interactive rates.

A summary of achievements is outlined in section 6.1 and suggestions for further work are given in 6.2.

6.1 Achievements

The main achievements of this thesis are:

- An introduction to the developing field of volume visualisation and volume graphics with a detailed review of existing volumetric techniques.
- A succinct review of volume rendering strategies for commodity graphics hardware which includes some improvements
- The introduction of a new image-order single pass direct volume rendering technique computed entirely on graphics hardware
- The development of a flexible object-oriented volume rendering platform
- The introduction of interactive generation and application of procedural textures for solid texturing and hypertexturing effects with a flexible architecture
- The introduction of interactive fine surface detail methods for volume representations
- The introduction of a new volume displacement mapping technique
- The introduction of mesostructure for volume datasets which removes the restriction of displaced surfaces requiring connection to the original surface.

6.2 Further Work

The graphics hardware manufacturers have recently released details of the upcoming graphics hardware in design and early production. These new graphics hardware accelerators include further optimisations and improvements, and will also introduce a new pipeline stage. This pipeline stage will be capable of defining the build up of geometry instead of simply working with vertices. This increased control will allow an efficient Marching Cubes or Marching Tetrahedral algorithm to be implemented entirely in hardware.

Additionally this new hardware should suit a combined object-order and image-order approach. An octree structure may be uploaded to the graphics hardware and recursively sampled with the ability to generate geometry for the fragment stage of the pipeline. This should allow techniques such as Multiple Pass and Single Pass direct volume rendering to benefit from an additional culling of fragments before any processing takes place. Similar methods have been proposed but fall short of an entirely efficient method.

The rate at which the algorithms in this thesis are executed will increase with each new generation of hardware, and the fundamentals are expected to remain fixed, however with an increased accuracy within texturing and blending stages of the pipeline some of the outlined speed-up methods can be successfully included without a loss of image quality.

Focus is generally given to rendering singular volume datasets, It is important to define mechanisms to render multiple datasets at the same time such as CVG. The increased horsepower of graphics accelerators will allow volumetric scene graphs to be rendered in real time. Additional strategies must be established concerning upload of expensive volume datasets to the graphics hardware for rendering in a multi-volume environment, additionally level of detail data storage and rendering will undoubtedly play an important role. The work presented in this thesis can play an important role in modelling and its inclusion with CVG would be ultimately beneficial. Voxelisation into triangle meshes might also benefit from the surface detail algorithms explored. A complex polygonal mesh can be voxelised into a distance field with its bump or displacement maps defining the high frequency detail in the resulting distance field.

It is further expected that ray-tracing and ultra fine detail rendering of volumetric scene graphs will be restricted to powerful computers, although it is possible that graphics hardware can accelerate stages of this process and thus global illumination approximation techniques must be employed for interactive rates.

There is still work to be carried out in finding the most efficient volume rendering platform on GPU hardware. All of the volume rendering algorithms explored suffer two problems.

The first problem is that they are entirely fragment shader bound, and the vertex shader is often idle during execution. This can be circumvented by employing the vertex shader to perform empty space leaping on future GPU's by affecting the proxy geometry used for ray casting or sample positions. Currently there is no method to generate a vertex list through the GPU pipeline. A future algorithm can utilise the vertex shader to perform hierarchical generation of proxy geometries such that there are less fragments to consider by analysing an object-order structure such as a min-max octree. Ray-casting can then be performed on the proxy geometries output by this stage of the pipeline which will result in a balanced pipeline with less work being computed entirely in the fragment shading stage of the pipeline.

The second problem is the current hardware implementations. The need for dynamic conditional expressions and in some cases loops introduce a drastic decrease in performance by introducing more cycles. Whilst methods such as multiple-pass rendering take the burden of these instructions away, they still require a heavy overhead in terms of memory and buffer swapping. Additionally only the IOS method can blend at full 32 bit precision. Other techniques would benefit from a truly 32 bit pipeline throughout the whole GPU architecture from 32 bit texture filtering to 32 bit blending. Currently increasing precision will decrease performance, however a full 32 bit pipeline should avoid these issues. A full 32 bit texture definition to include filtering will be important in future high-quality volume rendering applications which can rely on accurate lookup tables to increase rendering speeds.

The notion of soft-object's is clearly a powerful tool in volume graphics to define a natural looking object. This single volume dataset modelling technique could therefore be imported

into the CVG operations over scalar fields to allow an intuitive modelling primitive. The soft-region's roll in animation is also clearly important for general graphics scenes and the time varying nature of such a CVG operation should also be represented.

The notion of a complete shading tree volume graphics language can be highlighted and previous work such as *vlib* provide many of these features in software. However a real-time hardware accelerated environment to allow modelling manipulation and animation would benefit from the techniques presented in this chapter as they are especially suited to a framework implementation.

An investigation into computing a volume dataset from an original height field is required to remove the undersampling of sharp edges in displacement mapping. Since the volume displacement mapping methods can utilise slab sampling, these under-sampled sharp edges are expected to be improved. Displacement mapping can be a useful tool in modelling as well as rendering. A subset of objects are described more intuitively by wrapping a displacement map around their existing structure. This removes the burden of having multiple datasets in a tree structure to define an object. Further study should be given to this modelling application in terms of existing techniques and strategies. The voxelisation of distance fields from triangular meshes with displacement maps is also an unexplored notion for rendering complex surface meshes with mesostructure.

Bibliography

- [Ake93] Kurt Akeley. Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, New York, NY, USA, 1993. ACM Press.
- [ARC05] A. Abdul-Rahman and M. Chen. Spectral volume rendering based on the kubelka-munk theory. *Computer Graphics Forum*, 24(3):413–422, 2005.
- [ASK94] Ricardo S. Avila, Lisa M. Sobierajski, and Arie E. Kaufman. Visualizing nerve cells. *IEEE Computer Graphics and Applications*, 14(5):11–13, 1994.
- [BHP46] F. Bloch, W.W. Hansen, and M. Packard. The nuclear induction experiment. *Physical Review*, 70:474–485, 1946.
- [BJNN97] Martin Brady, Kenneth Jung, H. T. Nguyen, and Thinh Nguyen. Two-phase perspective ray casting for interactive volume navigation. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 183–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.
- [Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.
- [Bli78a] James F. Blinn. *Computer display of curved surfaces*. PhD thesis, University of Utah, 1978.
- [Bli78b] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292. ACM Press, 1978.
- [Bli82] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 21–29, New York, NY, USA, 1982. ACM Press.
- [Bli94] James F. Blinn. Image compositing–theory. *IEEE Computer Graphics and Applications*, 14(5):83–87, Sept 1994.
- [Blo88] J. Bloomenthal. Polygonization of implicit surfaces. *Compututer Aided Geometry and Design*, 5(4):341–355, 1988.

- [BM93] Barry G. Becker and Nelson L. Max. Smooth transitions between bump rendering algorithms. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 183–190, New York, NY, USA, 1993. ACM Press.
- [BMW98] David E. Breen, Sean Mauch, and Ross T. Whitaker. 3d scan conversion of csg models into distance volumes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 7–14, New York, NY, USA, 1998. ACM Press.
- [BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.
- [BPS97] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. The contour spectrum. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 167–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.
- [BR98] Uwe Behrens and Ralf Ratering. Adding shadows to a texture-based volume renderer. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 39–46, New York, NY, USA, 1998. ACM Press.
- [BS86] E. Bier and K. Sloan. Two-part texture mapping. *IEEE Computer Graphics and Applications*, 6(9):40–53, 1986.
- [Cat74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.
- [Cat75] Edwin E. Catmull. Computer display of curved surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, May 1975.
- [CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, pages 91–98. ACM Press, 1994.
- [CH02] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131, 2002.
- [Che95] E. Chernyaev. Marching cubes 33: Construction of topologically correct iso-surfaces. Technical Report CERN CN 95-17, 1995.
- [Che05] Min Chen. Combining point clouds and volume objects in volume scene graphs. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 127–135, 2005.
- [CN94] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, University of North Carolina at Chapel Hill, 1994.

- [Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.
- [Cor06] NVIDIA Corporation. Implementing the fixed function pipeline in cg. http://developer.nvidia.com/object/cg_fixed_function.html, 09 2006.
- [CRZP04] Wei Chen, Liu Ren, Matthias Zwicker, and Hanspeter Pfister. Hardware-accelerated adaptive EWA volume splatting. In *Proceedings of IEEE Visualization 2004*, October 2004.
- [CS78] H. N. Christiansen and T. W. Sederberg. Conversion of complex contour line definitions into polygonal element mosaics. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 187–192, New York, NY, USA, 1978. ACM Press.
- [CSW⁺03] M. Chen, D. Silver, A. S. Winter, V. Singh, and N. Cornea. Spatial transfer functions: a unified approach to specifying deformation in volume modeling and animation. In *VG '03: Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 35–44, New York, NY, USA, 2003. ACM Press.
- [CT00] Min Chen and John V. Tucker. Constructive volume geometry. *Computer Graphics Forum*, 19(4):281–293, December 2000.
- [DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1988. ACM Press.
- [DKC⁺98] Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, and Arie Kaufman. High-quality volume rendering using texture mapping hardware. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 69–ff., New York, NY, USA, 1998. ACM Press.
- [Dür88] M. Dürst. Letters: Additional references on marching cubes. *Computer Graphics*, 22(2):72–73, 1988.
- [EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM Press, 2001.
- [EMP⁺03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modelling: A Procedural Approach*. Morgan Kaufmann, 3rd edition, 2003.
- [FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

- [FKU77] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, 1977.
- [FL78] K. S. Fu and S. Y. Lu. Computer generation of texture using a syntactic approach. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 147–152, New York, NY, USA, 1978. ACM Press.
- [FSV98] S. Fang, R. Srinivasan, and S. Venkataraman. Volumetric csg? a model-based volume visualization approach. In *Sixth International Conference in Central Europe on Computer Graphics and Visualization*, 1998.
- [FvDFH96] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley, 2nd edition edition, 1996.
- [Gib97] Sarah F. Gibson. 3d chainmail: a fast algorithm for deforming volumetric objects. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 149–ff., New York, NY, USA, 1997. ACM Press.
- [GK96] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *Proceedings of the 1996 symposium on Volume visualization*, pages 23–ff. IEEE Press, 1996.
- [Gou71] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20(6):623–629, June 1971.
- [GPRJ00] Sarah F. Frisken Gibson, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000*, pages 249–254, 2000.
- [Gre05] Simon Green. *Implementing Improved Perlin Noise*, chapter 26, pages 409–416. Addison Wesley, 2005.
- [GS99] Nikhil Gagvani and Deborah Silver. Parameter-controlled volume thinning. *CVGIP: Graphical Models and Image Processing*, 61(3):149–164, 1999.
- [GS01] Nikhil Gagvani and Deborah Silver. Animating volumetric models. *Graphical Models*, 63(6):443–458, 2001.
- [GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaille. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 213–222, New York, NY, USA, 1984. ACM Press.
- [Har01] John C. Hart. Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–94. ACM Press, 2001.
- [HB86] K. H. Höhne and R. Bernstien. Shading 3d images from ct using grey level gradients. *IEEE Transactions on Medical Imaging*, 5(1):45–47, March 1986.

- [HCK⁺99] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts, and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 45–53, New York, NY, USA, 1999. ACM Press.
- [HDD⁺93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 19–26, New York, NY, USA, 1993. ACM Press.
- [HE99] Matthias Hopf and Thomas Ertl. Accelerating 3d convolution using graphics hardware (case study). In *VIS '99: Proceedings of the conference on Visualization '99*, pages 471–474, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [Hea] GE Healthcare. <http://www.gehealthcare.com/us/en/ct/products/ebt.html>.
- [HEGD04] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.
- [HHKP96] Taosong He, Lichan Hong, Arie Kaufman, and Hanspeter Pfister. Generation of transfer functions with stochastic search techniques. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 227–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.
- [HHM03] Markus Hadwiger, Helwig Hauser, and Torsten Möller. Quality issues of hardware-accelerated high-quality filtering on pc graphics hardware. *Journal of WSCG*, 11:213–210, 2003.
- [HL90] Pat Hanrahan and Jimi Lawson. A language for shading and lighting calculations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298, New York, NY, USA, 1990. ACM Press.
- [HMS95] Wolfgang Heidrich, Michael McCool, and John Stevens. Interactive maximum projection volume rendering. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 11, Washington, DC, USA, 1995. IEEE Computer Society.
- [Hou72] G. Hounsfield. A method of an apparatus for examination of a body by radiation such as x-ray or gamma radiation. Patent Specification 1283915, 1972.
- [HQB05] Wei Hong, Feng Qiu, and Arie Kaufman. Gpu-based object-order ray-casting for large datasets. In *Proceedings of the International Workshop on Volume Graphics '05*, 2005.

- [HSH04] Knud Henriksen, Jon Sporring, and Kasper Hornbaek. Virtual trackballs revisited. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):206–216, 2004.
- [HTF97] Bernd Hamann, Issac J. Trotts, and Gerald E. Farin. On approximating contours of the piecewise trilinear interpolant using triangular rational-quadratic bezier patches. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):215–227, 1997.
- [HTG01] Markus Hadwiger, Thomas Theuß, and Helwig Hauser and Eduard Gröller. Hardware-accelerated high-quality filtering on pc graphics hardware. In *Vision, Modeling, and Visualization '02*, 2001.
- [HVTH02] Markus Hadwiger, Ivan Viola, Thomas Theußl, and Helwig Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Vision, Modeling, and Visualization '02*, 2002.
- [HWK94] Taosong He, Sidney Wang, and Arie Kaufman. Wavelet-based volume morphing. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 85–92, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [IDSC04] Shoukat Islam, Swapnil Dipankar, Deborah Silver, and Min Chen. Spatial and temporal splitting of scalar fields in volume graphics. In *IEEE Symposium on Volume Visualization and Graphics (VV'04)*, 2004.
- [ILGS03] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 61, Washington, DC, USA, 2003. IEEE Computer Society.
- [JBŠ06] Mark W. Jones, Andreas Bærentzen, and Miloš Šrámek. Discrete 3D distance fields: Techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, July/August 2006.
- [JC94] M.W. Jones and M. Chen. A new approach to the construction of surfaces from contour data. *Computer Graphics Forum*, 13(4):75–84, 1994.
- [JDR04] Robert Jagnow, Julie Dorsey, and Holly Rushmeier. Stereological techniques for solid textures. *ACM Transactions on Graphics*, 23(3):329–335, 2004.
- [Jon96] Mark W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, 1996.
- [Kau04] Jan Kautz. Hardware lighting and shading: a survey. *Computer Graphics Forum*, 23(1):85–112, 2004.
- [KBR] John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language. <http://oss.sgi.com/projects/ogl-sample/registry/ARB>.
- [KCY93] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, 1993.

- [KD98] Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86, New York, NY, USA, 1998. ACM Press.
- [Kep75] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.
- [KF05] Emmett Kilgariff and Randima Fernando. *GPU Gems 2*, chapter The GeForce 6 Series GPU Architecture, pages 471–491. Addison Wesley, 2005.
- [KH84] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1984. ACM Press.
- [Kil96] Mark J. Kilgard. *Programming OpenGL for the X Windows System*, chapter 4. Addison-Wesley, 1996.
- [Kil00] M. J. Kilgard. A practical and robust bump-mapping technique for today's gpus. In *Game Developers Conference*, July 2000.
- [KKH01] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 255–262, Washington, DC, USA, 2001. IEEE Computer Society.
- [KKH02] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.
- [KPHE02] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of the conference on Visualization '02*, pages 109–116. IEEE Computer Society, 2002.
- [KR88] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.
- [KW03] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.
- [LB03] Adriano Lopes and Ken Brodlie. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):16–29, 2003.
- [LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

- [Lev88] Mark Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.
- [Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.
- [Lew89] J. P. Lewis. Algorithms for solid noise synthesis. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 263–270, New York, NY, USA, 1989. ACM Press.
- [LGL95] Apostolos Leros, Chase D. Garfinkle, and Marc Levoy. Feature-based volume metamorphosis. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 449–456, New York, NY, USA, 1995. ACM Press.
- [LHJ99] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 355–361, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458. ACM Press, 1994.
- [LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.
- [LWM04] Eric Lum, Brett Wilson, and Kwan-Liu Ma. High-quality lighting and efficient pre-integration for volume rendering. In *Proceedings of the Joint Eurographics-IEEE TVCG Symposium on Visualization 2004*, 2004.
- [Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.
- [MB94] Nelson L. Max and Barry G. Becker. Bump shading for volume textures. *IEEE Computer Graphics and Applications*, 14(4):18–20, 1994.
- [ME05] Ben Mora and David Ebert. Low-complexity maximum intensity projection. *ACM Transactions on Graphics*, 24(5):1392–1416, October 2005.
- [Mea82] D. Meagher. Geometric modelling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.
- [MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.
- [MHS99] Michael Meißner, Ulrich Hoffmann, and Wolfgang Straßer. Enabling classification and shading for 3d texture mapping based volume rendering using opengl and extensions. In *VIS '99: Proceedings of the conference on Visual-*

- ization '99, pages 207–214, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [Mic06] Microsoft. Directx, 9 2006.
- [MJ05] C. M. Miller and M. W. Jones. Texturing and hypertexturing of volumetric objects. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 117–125, 2005.
- [MJC02] Benjamin Mora, Jean-Pierre Jessel, and René Caubet. A new object-order ray-casting algorithm. In *IEEE Visualization '02*, pages 203–210, October 2002.
- [MMK⁺98] Torsten Möller, Klaus Mueller, Yair Kurzion, Raghu Machiraju, and Roni Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 143–151, New York, NY, USA, 1998. ACM Press.
- [MMMY96] Torsten Möller, Raghu Machiraju, Klaus Mueller, and Roni Yagel. Classification and local error estimation of interpolation and derivative filters for volume rendering. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 71–ff., Piscataway, NJ, USA, 1996. IEEE Press.
- [MMMY97] Torsten Möller, Raghu Machiraju, Klaus Mueller, and Roni Yagel. Evaluation and design of filters using a taylor series expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–199, 1997.
- [MN88] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer-graphics. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 221–228, New York, NY, USA, 1988. ACM Press.
- [Nat94] B. K. Natarajan. On generating topologically consistent isosurfaces from uniform samples. *The Visual Computer*, 11(1):52–62, 1994.
- [NH91] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Visualization 91*, pages 83–90, 1991.
- [Nie03] Gregory M. Nielson. On marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):283–297, 2003.
- [NM01] Manjushree Nulkar and Klaus Mueller. Splatting with shadows. In *Proceedings of the International Workshop on Volume Graphics '01*, 2001.
- [NS97] Gregory M. Nielson and Junwon Sung. Interval volume tetrahedrization. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 221–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.
- [Ope] OpenGL. The opengl architecture review board. <http://oss.sgi.com/projects/ogl-sample/registry/>.
- [PAC97] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *SIGGRAPH '97: Proceedings of the 24th annual conference on*

- Computer graphics and interactive techniques*, pages 303–306, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.
- [PASS95] Alexander A. Pasko, Valery Adzhiev, Alexei Sourin, and Vladimir V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.
- [PBFJ05] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. *ACM Transactions on Graphics*, 24(3):626–633, 2005.
- [PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.
- [PD84] Thomas Porter and Tom Duff. Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 253–259, New York, NY, USA, 1984. ACM Press.
- [Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 279–286. ACM Press, 1985.
- [Per85] Ken Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.
- [Per02] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682. ACM Press, 2002.
- [PH89] K. Perlin and E. M. Hoffert. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262. ACM Press, 1989.
- [PHK⁺99] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 251–260, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.
- [Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.
- [PLB⁺01] Hanspeter Pfister, Bill Lorensen, Chandrajit Bajaj, Gordon Kindlmann, Will Schroeder, Lisa S. Avila, Ken Martin, Raghu Machiraju, and Jinho Lee. The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21(3):16–22, 2001.
- [Pro] The Visible Human Project. <http://www.nlm.nih.gov/research/visible/>.

- [PT90] B. A. Payne and A. W. Toga. Surface mapping brain function on 3d models. *IEEE Computer Graphics and Applications*, 10(5):33–41, September 1990.
- [PTP45] E.M. Purcell, H.C. Torrey, , and R.V. Pound. Resonance absorption by nuclear magnetic moments in a solid. *Physical Review*, 69:37, 1945.
- [Req80] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.
- [RGW⁺03] Stefan Röttger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the symposium on Data visualisation 2003*, pages 231–238. Eurographics Association, 2003.
- [RSEB⁺00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM Press, 2000.
- [RTB⁺92] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann, and Amitabh Varshney. Real-time procedural textures. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 95–100, New York, NY, USA, 1992. ACM Press.
- [RV77] A.A.G. Requicha and G.B. Voelcker. Constructive solid geometry. Technical report, University of Rochester,, 1977.
- [SA] Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 1.5).
- [Sab88] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 51–58, New York, NY, USA, 1988. ACM Press.
- [Sat01] Richard Satherley. *Computation and Application of Distance Fields in Volume Graphics*. PhD thesis, University of Wales, Swansea, 2001.
- [SB02] S. Svensson and G. Borgefors. Digital distance transforms in 3D images using information from neighbourhoods up to $5 \times 5 \times 5$. *Computer Vision and Image Understanding*, 88:24–53, 2002.
- [SJ01] R. Satherley and M. W. Jones. Vector-city vector distance transform. *Computer Vision and Image Understanding*, 82(3):238–254, 2001.
- [SJ02] R. Satherley and M. W. Jones. Hypertexturing complex volume objects. *The Visual Computer*, 18(4):226–235, June 2002.
- [SK90] D. Speray and S. Kennon. Volume probes: Interactive data exploration on arbitrary grids. *Computer Graphics*, 24(5):5–12, November 1990.

- [SK94] Lisa M. Sobierajski and Arie E. Kaufman. Volumetric ray tracing. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 11–18, New York, NY, USA, 1994. ACM Press.
- [ŠK00] Miloš Šrámek and Arie Kaufman. Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):236–252, 2000.
- [Šrá96] Miloš Šrámek. Fast ray-tracing of rectilinear volume data. In *Proceedings of the Eurographics workshop on Virtual environments and scientific visualization '96*, pages 201–210, London, UK, 1996. Springer-Verlag.
- [SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.
- [SW05] P. Shen and P. Willis. Texture mapping volume objects. In *Vision, Video, and Graphics*, pages 45–52, 2005.
- [SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 65–70, New York, NY, USA, 1992. ACM Press.
- [THB⁺90] Ulf Tiede, Karl Heinz Hoehne, Michael Bomans, Andreas Pommert, Martin Riemer, and Gunnar Wiebecke. Surface rendering: Investigation of medical 3d-rendering algorithms. *IEEE Computer Graphics and Applications*, 10(2):41–53, 1990.
- [THR97] U. Tiede, K.H. Hohne, and M. Riemer. Comparison of surface rendering techniques for 3d tomographic objects. In *Computer Assisted Radiology*, pages 599–610, 1997.
- [TLM03] Fan-Yin Tzeng, Eric B. Lum, and Kwan-Liu Ma. A novel interface for higher-dimensional classification of volume data. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 66, Washington, DC, USA, 2003. IEEE Computer Society.
- [Tur92] Greg Turk. Re-tiling polygonal surfaces. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 55–64, New York, NY, USA, 1992. ACM Press.
- [Tze05] Fan-Yin Tzeng. An intelligent system approach to higher-dimensional classification of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):273–284, 2005. Member-Eric B. Lum and Senior Member-Kwan-Liu Ma.
- [UK88] Craig Upson and Michael Keeler. V-buffer: visible volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer*

- graphics and interactive techniques*, pages 59–64, New York, NY, USA, 1988. ACM Press.
- [War91] G. Ward. *Graphic Gems II*, chapter A recursive Implementation of the Perlin Noise Function, pages 396–401. Academic Press Professional, 1991.
- [WC01] A.S. Winter and M. Chen. Vlib: A volume graphics api. In *International workshop on Volume Graphics '01*. Springer-Wien New York, 2001.
- [WE98a] R. Westermann and T. Ertl. Solid texturing on a per-pixel basis. In *IEEE Multidimensional Digital Signal Processing '98, Conference Proceedings*, pages 48–55. IEEE, 1998.
- [WE98b] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA, 1998. ACM Press.
- [WEE02] Daniel Weiskopf, Klaus Engel, and Thomas Ertl. Volume clipping via per-fragment operations in texture-based volume visualization. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 93–100, Washington, DC, USA, 2002. IEEE Computer Society.
- [WEE03] Daniel Weiskopf, Klaus Engel, and Thomas Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.
- [Wes90] Lee Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA, 1990. ACM Press.
- [WG90] Jane Wilhelms and Allen Van Gelder. Topological considerations in isosurface generation extended abstract. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 79–86, New York, NY, USA, 1990. ACM Press.
- [WG92] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.
- [Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.
- [Wil83] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, 1983.
- [Win02] Andrew Winter. *Volume Graphics: Field Based Modelling and Rendering*. PhD thesis, University of Wales, Swansea, 2002.
- [WJ06] Simon J. Walton and Mark W. Jones. Volume wires : A framework for empirical non-linear deformation of volumetric datasets. *Journal of WSCG*, 14:81–88, 2006.

- [WK93] Sidney W. Wang and Arie E. Kaufman. Volume sampled voxelization of geometric primitives. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 78–84, 1993.
- [WMG98] Craig M. Wittenbrink, Thomas Malzbender, and Michael E. Goss. Opacity-weighted color interpolation, for volume sampling. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 135–142, New York, NY, USA, 1998. ACM Press.
- [WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, February 1986.
- [WS01] Rüdiger Westermann and Bernd Sevenich. Accelerated volume ray-casting using texture mapping. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 271–278, Washington, DC, USA, 2001. IEEE Computer Society.
- [WTL⁺04] Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. Generalized displacement maps. In *Eurographics Symposium on Rendering*, 2004.
- [WVW94] Orion Wilson, Allen VanGelder, and Jane Wilhelms. Direct volume rendering via 3d textures. Technical report, University of California, Santa Cruz, Santa Cruz, CA, USA, 1994.
- [WWT⁺03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, 2003.
- [YCK92] R Yagel, D Cohen, and A Kaufman. Discrete ray tracing. *IEEE Computer Graphics and Applications*, 12(9):19–28, September 1992.
- [YK92] Roni Yagel and Arie Kaufman. Template-based volume viewing. *Computer Graphics Forum*, 11(3):153–167, 1992.
- [YKI03] Shuntaro Yamazaki, Kiwamu Kase, and Katsushi Ikeuchi. Hardware-accelerated visualization of volume-sampled distance fields. In *Proceedings of the Shape Modeling International 2003*, page 264. IEEE Computer Society, 2003.
- [YKZ91] Roni Yagel, Arie Kaufman, and Qiang Zhang. Realistic volume imaging. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 226–231, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
- [ZC02] Caixia Zhang and Roger Crawfis. Volumetric shadows using splatting. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 85–92, Washington, DC, USA, 2002. IEEE Computer Society.
- [ZJ91] C. Zuhlten and H. Jurgens. Continuation methods for approximating isovalued complex surfaces. In *Computer Graphics Forum, Proc. Eurographics '91*, pages 5–19, September 1991.

-
- [ZPvBG01] M. Zwicker, H. Pfister, J. van Baar, and M. H. Gross. Ewa volume splatting. In *IEEE Visualization '01*, 2001.
- [ZPvG02] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

List of Figures

2.1	Differing grid geometries and topologies for volume data	7
2.2	Modern CT Scanner, GE eSpeed EBT [Hea]	8
2.3	Axial images of CT, MRI and colour cryosections of the head	9
2.4	Relationship between a voxel and a cell (regular grid)	10
2.5	1D Reconstruction filters	11
2.6	Trilinear interpolation in a volume cell	12
2.7	Differing classifications of the <i>CTHead</i> dataset	16
2.8	Normal vectors defined for Gouraud and Phong shading models	19
2.9	Marching cubes vertex enumeration scheme	21
2.10	Marching cubes basic case table	22
2.11	Marching tetrahedra basic case table	23
2.12	<i>AVSHydrogen</i> dataset images from the marching tetrahedra algorithm	24
2.13	Octree data structure overview	25
2.14	MIP rendering of the <i>CTHead</i>	29
2.15	Ray-casting overview	29
2.16	Equidistant ray sampling during ray-casting	30
2.17	2D ray-casting overview	31
2.18	Solid texture block and solid textured object	40
2.19	Sphere with object density function defining a soft-region.	43
2.20	Bias curve functions	44
2.21	Gain curve functions	44
3.1	GPU pipeline overview	50
3.2	Generic hardware pipeline	51
3.3	Vertex shader execution algorithm	55
3.4	Fragment shader execution algorithm	57
3.5	2D Mip-map texture	58
3.6	Differing proxy slice strategies	61
3.7	Image and object aligned proxy geometries	63
3.8	Spherical shell proxy geometry	66
3.9	Distance field rendering: Empty space leaping along a ray	71
3.10	Distance field rendering: Adaptively rendering along a ray	72
3.11	Distance field ray samples through volume	72
3.12	Slab encoded between two proxy slices	75
3.13	Pre-integrated transfer function table	77

3.14	1D Transfer function	77
3.15	Iso-surface lookup tables for slab rendering	77
3.16	Multiple iso-surfaces of the <i>SphereDist</i> dataset	78
3.17	Lighting discontinuity in slab rendering for interpolated gradients	79
3.18	Screenshot of software testing environment	81
3.19	Image aligned proxy geometry generation	83
3.20	OOP vertex shader	84
3.21	OOP post-classification fragment shader	85
3.22	OOP pre-integrated classification fragment shader	85
3.23	OOP lit post-classification fragment shader	85
3.24	OOP lit pre-integrated fragment shader	86
3.25	OOP iso-surface fragment shader using conditionals	86
3.26	OOP iso-surface fragment shader for alpha test method	86
3.27	OOP multiple interpolated iso-surface fragment shader	87
3.28	<i>BuckyBall</i> dataset post-classification and pre-integrated classification images	89
3.29	<i>BuckyBall</i> dataset iso-surface and interpolated iso-surface images	90
3.30	<i>CTHead</i> dataset post-classification and pre-integrated classification images	92
3.31	<i>CTHead</i> dataset iso-surface and interpolated iso-surface images	93
3.32	IOM proxy geometry	95
3.33	Ping-pong rendering scheme	96
3.34	IOM vertex shader	96
3.35	IOM post-classification fragment shader	98
3.36	IOM pre-integrated classification fragment shader	99
3.37	IOM iso-surface fragment shader	100
3.38	IOM interpolated iso-surface fragment shader	101
3.39	IOS post-classification fragment shader	105
3.40	IOS pre-integrated classification fragment shader	106
3.41	IOS isosurface fragment shader	107
3.42	IOS interpolated isosurface fragment shader	107
4.1	Examples of Perlin noise	114
4.2	Examples of noise implementations	117
4.3	Examples of turbulence implementations	118
4.4	OOP solid texturing vertex shader	122
4.5	OOP solid texturing fragment shader	122
4.6	OOP interpolated solid texturing fragment shader	123
4.7	IOS solid texturing fragment shader	123
4.8	IOS interpolated solid texturing fragment shader	124
4.9	<i>BuckyBall</i> dataset bonzo solid texture images	126
4.10	<i>CTHeadDist</i> dataset turbulence solid texture images	127
4.11	<i>SphereDist</i> dataset wood solid texture images	128
4.12	<i>CTHeadDist</i> dataset with object density function defining a soft-region.	132
4.13	Bias and Gain lookup tables	134
4.14	Examples of hypertexture applied to the <i>SphereDist</i> dataset	137
4.15	OOP hypertexture fragment shader	138
4.16	OOP hypertexture and iso-surface fragment shader	138

4.17	OOP interploated hypertexture fragment shader	139
4.18	OOP interploated hypertexture and iso-surface fragment shader	139
4.19	IOS hypertexture fragment shader	140
4.20	IOS hypertexture and iso-surface fragment shader	140
4.21	IOS interpolated hypertexture fragment shader	141
4.22	IOS interpolated hypertexture and iso-surface fragment shader	142
4.23	<i>SphereDist</i> dataset fur hypertexture and interpolated hypertexture images	145
4.24	<i>CTHeadDist</i> dataset fireball hypertexture and interpolated hypertexture	146
4.25	Animation of PerlinFire at differing time intervals	150
5.1	Mapping strategies for O and O^{-1}	155
5.2	Painting on the surface of the <i>BuckyBall</i> dataset	156
5.3	Intermediate surface geometries	158
5.4	OOP 2D texturing fragment shader	161
5.5	OOP interpolated 2D texturing fragment shader	161
5.6	IOS 2D texturing fragment shader	162
5.7	IOS interpolated 2D texturing fragment shader	162
5.8	<i>BuckyBall</i> dataset 2D texture mapping and interpolated 2D texture mapping	164
5.9	<i>CTHeadDist</i> 2D texture mapping and interpolated 2D texture mapping	165
5.10	Bump mapping stages	169
5.11	3D Bump mapping normal perturbation	170
5.12	OOP 3D bump mapping fragment shader	171
5.13	OOP 3D interpolated bump mapping fragment shader	171
5.14	IOS 3D bump mapping fragment shader	172
5.15	IOS 3D interpolated bump mapping fragment shader	173
5.16	<i>BuckyBall</i> dataset 3D bump mapping and interpolated 3D bump mapping	174
5.17	<i>CTHeadDist</i> 3D bump mapping and interpolated 3D bump mapping	175
5.18	Normal mapping for 2D bump mapping and 2D displacement mapping	177
5.19	OOP 2D bump mapping fragment shader	178
5.20	OOP 2D interpolated bump mapping fragment shader	178
5.21	IOS 2D bump mapping fragment shader	179
5.22	IOS 2D interpolated bump mapping fragment shader	180
5.23	<i>BuckyBall</i> dataset 2D bump mapping and interpolated 2D bump mapping	181
5.24	<i>CTHeadDist</i> 2D bump mapping and interpolated 2D bump mapping	182
5.25	2D and 3D displacement mapping strategies	185
5.26	OOP 2D displacement mapping fragment shader	187
5.27	IOS 2D displacement mapping fragment shader	188
5.28	<i>CTHeadDist</i> dataset 2D displacement mapping images	189
5.29	OOP volume displacement mapping fragment shader	191
5.30	OOP volume interpolated displacement mapping fragment shader	192
5.31	IOS volume displacement mapping fragment shader	193
5.32	IOS volume interpolated displacement mapping fragment shader	194
5.33	<i>SphereDist</i> volume displacement mapping	195

List of Tables

1.1	Featured Datasets	4
3.1	GPU generations and additional hardware capabilities	52
3.2	Example GPU series	53
3.3	GPU generation texture unit capabilities	59
3.4	GPU Branching Costs	60
3.5	Rendering approach and segmentation method comparison matrix	82
3.6	OOP rendering frame rates	88
3.7	IOM frame rates	102
3.8	IOS frame rates	108
4.1	GPU Noise implementation comparisons	119
4.2	OOP solid texturing frame rates	129
4.3	IOS solid texturing frame rates	130
4.4	IOS solid texturing frame rates with empty space leaping	131
4.5	OOP hypertexture frame rates	143
4.6	IOS hypertexture frame rates	147
4.7	IOS hypertexture frame rates with empty space leaping	148
5.1	2D texture mapping frame rates	166
5.2	3D bump mapping frame rates	173
5.3	2D bump mapping frame rates	180
5.4	2D displacement mapping frame rates	190
5.5	Volume displacement mapping frame rates	192